Elcock
& Michie

The Machine Intelligence Hypothesis

Ellis
Horwood

EDINBURGH
UNIVERSITY
LIBRARY

Author's Surname
Catalogue Heading MACHINE
INTELLIGENCE

Shelf Mark Machine Intelligence
~~& Perception~~ Library

13293.

27.10.80

£28.00

MACHINE INTELLIGENCE 8

Machine Representations of Knowledge

Machine Intelligence Volumes 1–7

Editor-in-Chief: Donald Michie

are all published by
Edinburgh University Press
and in the United States of America by
Halsted Press (a subsidiary of John Wiley & Sons, Inc.)

MACHINE INTELLIGENCE 8

edited by

E. W. ELCOCK
Professor and Chairman, Department of Computer Science
University of Western Ontario

and

DONALD MICHIE
Professor of Machine Intelligence
University of Edinburgh

ELLIS HORWOOD LIMITED
Publisher Chichester

Halsted Press: a division of
JOHN WILEY & SONS INC.
New York · London · Sydney · Toronto

The publisher's colophon is reproduced from James Gillison's drawing of the ancient Market Cross, Chichester

First published in 1977 by

ELLIS HORWOOD LIMITED
Coll House, Westergate, Chichester, Sussex, England

Distributors:

Australia, New Zealand, South-east Asia:
JOHN WILEY & SONS AUSTRALASIA PTY LIMITED
1-7 Waterloo Road, North Ryde, N.S.W., Australia

Canada:
JOHN WILEY & SONS CANADA LIMITED
22 Worcester Road, Rexdale, Ontario, Canada

Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester, Sussex, England

North and South America and the rest of the world:
HALSTED PRESS, a division of
JOHN WILEY & SONS INC.,
605 Third Avenue, New York, N.Y.10016, U.S.A.

© 1977 E. W. Elcock & D. Michie / Ellis Horwood Ltd

Library of Congress Catalog Card Number: 67-13648
ISBN (Ellis Horwood) 85312-058-7
ISBN (Halsted Press) 0-470-99059-7

Printed in Great Britain by Biddles of Guildford

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical photocopying, recording or otherwise, without prior permission.

CONTENTS

INTRODUCTION

KNOWLEDGE AND MATHEMATICAL REASONING

1 Representation of knowledge in a geometry machine. E.W. ELCOCK 11
2 Typing and proof by cases in program verification. W.W. BLEDSOE and M. TYSON 30

PROBLEM-SOLVING AND DEDUCTION

3 Practical and theoretical considerations in heuristic search algorithms. I. POHL 55
4 Variable range restrictions in resolution theorem proving. S. SICKEL 73
5 A formal model for space-filling puzzles. W.M. MCKEEMAN 86
6 Achieving several goals simultaneously. R. WALDINGER 94

MEASUREMENT OF KNOWLEDGE

7 Dynamic probability, computer chess, and the measurement of knowledge. I.J. GOOD 139
8 A theory of advice. D. MICHIE 151

INDUCTIVE ACQUISITION OF KNOWLEDGE

9 Rationality, evidence, and induction in scientific inference. I.J. GOOD 171
10 An experiment on inductive learning in chess end games. R.S. MICHALSKI and P.G. NEGRI 175
11 Inductive learning in a hierarchical model for representing knowledge in chess end games. P.G. NEGRI 193
12 Summing up of the discussion on inductive inference. I.J. GOOD 205

PROGRAMMING TOOLS FOR KNOWLEDGE-REPRESENTATION

13 Programming language design for the representation of knowledge. J.M. FOSTER 209
14 Some observations on conceptual programming. E. SANDEWALL 223
15 Programming with resolution logic. M.H. van EMDEN 266
16 An overview of production systems. R. DAVIS and J. KING 300

DIALOGUE-TRANSFER OF KNOWLEDGE TO MACHINES

17 A hypothetical dialogue exhibiting a knowledge base for a program-understanding system. C. GREEN and D. BARSTOW 335

DIALOGUE-TRANSFER OF KNOWLEDGE TO HUMANS

18 Representing knowledge about mathematics for computer-aided teaching, part I – educational applications of conceptualizations from artificial intelligence. R.B. DAVIS 363
19 Representing knowledge about mathematics for computer-aided teaching, part II – the diversity of roles that a computer can play in assisting learning. R.B. DAVIS, S. DUGDALE, D. KIBBEY, C. WEAVER 387
20 Three interactions between AI and education. K. KAHN 422

CASE STUDIES IN EMPIRICAL KNOWLEDGE

21 Knowledge representation for archaeological inference. J. DORAN 433
22 Chromosome classification and segmentation as exercises in knowing what to expect. D. RUTOVITZ 455

PERCEPTUAL KNOWLEDGE

23 A duality concept for the analysis of polyhedral scenes. D.A. HUFFMAN 475
24 Realizable configurations of lines in pictures of polyhedra. D.A. HUFFMAN 493
25 How to see a simple world: an exegesis of some computer programs for scene analysis. A.K. MACKWORTH 510

WORLD-KNOWLEDGE FOR LANGUAGE-UNDERSTANDING

26 Inference and knowledge in language comprehension. E. CHARNIAK 541
27 Representation and understanding of text. R. SCHANK 575

SUBJECT AND NAME INDEX 621

AUTHOR INDEX 627

INTRODUCTION

The eighth volume completes a ten-year span of the *Machine Intelligence* series. It is appropriate, therefore, to take stock of the main events, and to note certain solid steps and occasional forward leaps.

Leaps are normally preceded by some preparatory back-tracking. The uniform procedures of heuristic search and resolution theorem-proving which dominated the scene in 1965 cannot of themselves, as we now see, be developed into "the answer" to automatic problem-solving. This realisation has paved the way for machine-aided forays into non-trivial mathematics, as indicated in Bledsoe and Tyson's contribution to this volume. On the other side of the coin we have the startling and unexpected discovery by Kowalski, developed further here by van Emden, that first-order predicate calculus can be fashioned into a workmanlike programming language and a resolution strategy into an interpreter capable of processing the language with reasonable efficiency.

In the same ten years computer vision and computer understanding of natural language have been rescued from Cinderella status to become cynosures around which the brightest talents have gravitated. In vision the work of Huffman, Waltz and Clowes comes to mind; this volume contains two new contributions from Huffman of characteristic originality and precision. In language-understanding, the semantics-first approach first broke the log-jam of the "machine translation" epoch with Winograd's and Woods' studies. It is dynamically represented and extended here by Charniak and Schank.

This decade has also seen long-predicted successes by that school of Computer-Aided Instruction which would prefer to interpret the initials as standing for Children's Artificial Intelligence. A section of this book is devoted to reports from this expanding sector of the front.

Finally we must recognise that ten years ago machine intelligence was still a precocious foundling, with little going for it but energy and aspiration to compensate for a bad name and a disputed nature. Now all has been clarified. Not a single AI professional can be found who will not say that whatever this Protean field's name should be its nature has at least been firmly established: the use of machines and machine-oriented formalisms to study how knowledge may be represented, measured, transferred, and acquired *de novo*. This book constitutes the first integrated collection devoted to a theme which we now perceive to have been our sole proper business all along.

The outstanding enabler of the scientific event here recorded was Dr. Tilo Kester and the Scientific Affairs Division of NATO. Funds were granted for a two-week NATO Advanced Study Institute at Santa Cruz, where the University of California provided material and moral support, and a gracious setting. In this connexion Dr. David Huffman's distinguished inspiration deserves special mention. His good offices were, in particular, instrumental in attracting a supplementary grant from the Office of Naval Research to enable the proceedings to be broadened beyond the confines of the strict NATO ASI frame, thus re-creating, in total effect, something closer to the traditional Workshop style. Its overall success was assured by Dr. Sharon Sickel's tireless work as organiser and treasurer.

After a number of publishing vicissitudes Scott and Laurie Preece of Urbana, Illinois, undertook to typeset the book and to convert edited typescripts into completed paste-ups in three months. The reader can judge for himself the dedication and exemplary standards of craftsmanship with which this was done. Our thanks are due to Mr. Ellis Horwood, the U.K. publisher, for completing the process to the printing, binding, distribution and promotion of the finished book.

The beneficent genie of the *Machine Intelligence* series, Mr. Archibald Turnbull, once more hovered over events, not this time as publisher but as adviser and, at certain critical junctures, as a generous and resourceful enabler.

E.W. Elcock
Donald Michie
October 1976

KNOWLEDGE AND MATHEMATICAL REASONING

1

Representation of Knowledge in a Geometry Machine

E. W. Elcock

Department of Computer Science
University of Western Ontario

PART 1

In their book *Mathematics and Logic* Kac and Ulam (1971) comment:

> "The point of view as it has evolved through centuries is that one need not know what things *are* as long as one knows what *statements* about them one is *allowed* to make. Hilbert's famous *Grundlagen der Geometrie* begins with the sentence: 'Let there be three kinds of objects; the objects of the first kind shall be called "points", those of the second kind "lines", and those of third "planes"'. That is all, except that there follows a list of initial statements (axioms) that involve the words "point", "line" and "plane", and from which other statements involving those undefined words can now be deduced by logic alone. This permits geometry to be taught to a blind man and even to a computer!"

Leaving aside the attitude implicit in Kac & Ulam's use of the word 'even' in the phrase 'even to a computer', it has become clear that programs to prove theorems in first order axiomatic theories such as geometry, working in this 'blind' way, are unlikely to be successful. In parentheses, one might remark that mathematicians, however they express their proofs, usually do not construct them by working entirely within the formal syntactic system (i.e. blind).

What does it mean *not* to be blind? In the case of geometry, one of the ways would be to use a diagram in which 'points' and 'lines' referred to in the premises of a theorem to be proved are made concrete in a diagram, and predicates and functions of the theorem such as collinear, intersection, etc. are given their usual geometric interpretation and can be evaluated by procedures operating on the diagram. The actual points and lines made concrete in the diagram should, of course, be chosen so that the premises of the theorem to be proved are true in the diagram. Thus for the following theorem:

Premises:

ΔABC
M is the mid-point of segment BC
BD is the perpendicular from B to AM
CE is the perpendicular from C to AM.

To prove:

segment BD = segment CE.

an appropriate diagram could be that of Figure 1(a).

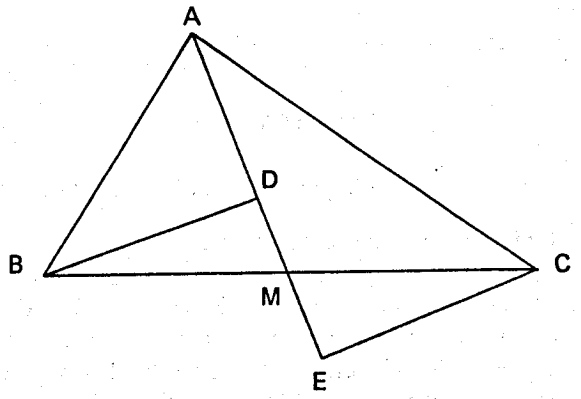


FIG. 1a

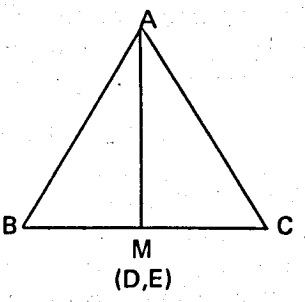


FIG. 1b

It is readily verified that the premises are true in the diagram. It is also worth remarking something that will be important later, namely that many things will be true in the diagrams which are *not* consequences of the premises of the theorem. Some of these, such as the fact that the length of segment AB is the particular multiple of the length of segment AC that is embodied in the diagram will usually be of no concern. Others might be. For example, if it were not for the premise

distinct (D,M,E),

a possible diagram would be that based on the isosceles triangle of Figure 1(b). Yet a "natural" illustrative diagram might still be taken to be that of Figure 1(a) when (potentially misleading) statements such as ΔBDM and ΔCEM are true in the diagram but are *not* implied by the (new) premises.

Despite an overabundance of things which are true in the diagram, for our purpose of proving theorems the diagram has a very important "inverse" property (to be stated more carefully later): anything which is *false* in the diagram is certainly *not* a consequence of the premises. Thus it is 'clear' in the diagram (in the sense of the processes underlying our perceptual comparison of angles) that ∠BAM ≠ ∠CAM. From what we have said, ∠BAM = ∠CAM cannot be a consequence of the premises (true in the diagram) of the theorem.

How is it that such a property of a diagram should be of great use in developing a proof of the associated theorem? Since this question motivates much of this paper, we will attempt an informal answer immediately. This we will do by suggesting the evolution and motivation of proof steps in the context of a simple example—the theorem stated earlier. So:

Premises:

ΔABC
M is the mid-point of segment BC
BD is the perpendicular from B to AM
CE is the perpendicular from C to AM
distinct (D,M,E)

Prove (G1):

segment BD = segment CE.

Informal proof:

As mentioned, we can draw a diagram (Figure 1(a)) to illustrate the theorem. We recall that there is a theorem of plane geometry which says that:

THEOREM 1: If two triangles are congruent, then their corresponding sides are equal.

We can clearly use this known theorem to prove that BD = CE if we can show that BD and CE are corresponding sides of two congruent triangles. The diagram suggests that we try:

Prove (G2):

ΔBDM = ΔCEM.

If we can prove G2, then Theorem 1 establishes our original goal G1 since BD

and CE are, indeed, corresponding sides of ΔBDM and ΔCEM. We shall see that the proof of G2 is straightforward but, before continuing the proof, let us pause and comment on the mechanisms which underly the apparent ease with which we, in fact, set up goal G2.

First, given that we have decided that we are going to use the tactic implicit in Theorem 1 and so attempt G2, how do we know that we can even assert ΔBDM and ΔCEM? We can say that it is 'obvious' (or can be 'assumed') from the diagram. More precisely, B, D and M are perceptually distinct and not collinear (the necessary and sufficient conditions for the assertion ΔBDM) in the diagram. The same is true of C, E and M.

Second, granted that we are going to choose a Δ with BM as a constituent side and a Δ with CM as a constituent side and try to prove them congruent, why did we choose the particular triangles ΔBDM and ΔCEM? Instead of G2 we could have set us any of the goals

$$\begin{aligned} &G2' \quad \Delta BDM \equiv \Delta CEA \\ &G2'' \quad \Delta BDA \equiv \Delta CEM \\ &G2''' \quad \Delta BDA \equiv \Delta CEA. \end{aligned}$$

Why did we not choose one of these for deeper (formal) exploration? It is suggested that these subgoals are not proposed for formal examination because in our diagram we can 'see' that these subgoals are patently false. Thus in the case of subgoal

$$\Delta BDA \equiv \Delta CEA:$$

we can 'see' that the necessary condition ∠ABD = ∠ACE is false where by the phrase 'see. . .is false' we again emphasize that we imply some evaluative procedure (computation) on the diagram. On the other hand ΔBDM and ΔCEM 'look' congruent where again we mean that evaluative checking procedures in the diagram succeed (e.g. ∠DBM = ∠ECM where the equality is in the framework of the visual procedures).

Finally, why choose a tactic based on Theorem 1 rather than some other? For example, with the particular initiating goal of proving two segments equal, we might have brought to bear tactics based on:

Theorem 1′: If ΔXYZ is such that its base angles ∠XYZ and ∠XZY are equal, then the sides XY and XZ opposite these angles are equal. (Tactic: to prove two segments equal, prove they are slant sides of a triangle whose base angles are equal), or

Theorem 1″: If segment XY = segment UV and segment RS = segment UV then segment XY = segment RS. (Transitivity of segment equality). (Tactic: to prove two segments equal, find a third segment which is equal to the original segments).

Again, the suggestion is that although such tactics might be tentatively con-

sidered as candidates for formal exploration, they are rejected on the basis of evaluative procedures in the diagram. Thus there is no triangle with equal base angles in the diagram, nor in the diagram is there a segment which is distinct from BD and CE and which appears to be equal to BD and CE.

This digression from our proof is motivated, as was mentioned in the introduction to the theorem to be proved, to show the important and many facetted role played by the diagram in a process of constructing a geometrical proof. For our expository purposes, the digressions are certainly more important than the emergent detailed proof, and soon we will want to consider both the formalisation and mechanisation of this role played by the diagram. Before doing this, however, let us complete the sketch of our proof: there are more insights still to be gained.

Our current goal is:

Prove (G2):

$\Delta BDM \equiv \Delta CEM$.

We now recall:

Theorem 2: If Δ's ΔXYZ and ΔRST are such that segment XZ = segment RT and $\angle XYZ = \angle RST$ and $\angle XZY = \angle RTS$ then the Δ's are congruent (i.e. the other pairs of corresponding sides and the other corresponding angles are each equal). (Tactic: if the goal is to prove that Δ's ΔXYZ and ΔRST are congruent, then try to prove the three goals

$$\begin{array}{c}\text{segment } XZ = \text{segment } RT \\ \angle XYZ = \angle RST \\ \angle XZY = \angle RTS \qquad)\end{array}$$

In the context of our goal

$$\Delta BDM \equiv \Delta CEM$$

considered in isolation (i.e. forgetting for the moment its motivational history), there are a number of instantiations of the general tactic. By an "instantiation" we refer to the process by which, in applying a theorem or tactic we have to say which concrete points in the diagram we are going to associate with (substitute for) the 'general' points of the theorem or tactic. Our earlier remark, concerned with choosing candidate triangle pairs for congruence, to the effect that ". . . on the other hand BDM and CEM 'look' congruent . . .", implies that our computational procedures on the diagram reject unsuitable associations such as:

$$X/B,\ Y/M,\ Z/D,\ R/E,\ S/C,\ T/M$$

which would lead to an attempt to prove the subgoal

$$\angle BMD = \angle ECM \text{ (instantiation of } \angle XYZ = \angle RST)$$

which is clearly false in the diagram. This is just another example of the kind of use of the diagram already explained. Rather different is the situation that arises if we try the association

$$X/B,\ Y/M,\ Z/D,\ R/C,\ S/M,\ T/E$$

which would lead to an attempt to prove the three subgoals

$$\text{segment } BD = \text{segment } CE$$
$$\angle BMD = \angle CME$$
$$\angle BDM = \angle CEM$$

none of which are obviously false in the diagram. In fact, the last two, of course, can be proven true ($\angle BMD = \angle CME$ since vertically opposite angles are equal, and $\angle BDM = \angle CEM$ since all right angles are equal), but the first subgoal is the original theorem we set out to prove! If we, or our geometry program, do not recognise this, we are in danger of repeating the proof path to this point over and over again indefinitely!

It is clear that our proof style leads to a proof structure which is a hierarchy (tree) of subgoals as in Figure 2. Each node represents a subgoal which is proved if its descendent subgoals can be proved. We must monitor that no node (subgoal) is identical to one of its ancestor nodes.

On the assumption that we avoid such pitfalls, let us briefly complete our proof. We are trying to prove G2:

$$\Delta BDM \equiv \Delta CEM$$

this in turn being motivated by G1:

$$\text{segment } BD = \text{segment } CE.$$

which together with other diagrammatic evidence suggests the appropriate instantiation of the tactic associated with theorem 2 is:

$$X/B,\ Y/D,\ Z/M,\ R/C,\ S/E,\ T/M$$

when the three subgoals to be proved to establish G2 are:

$$\text{segment } BM = \text{segment } CM$$
$$\angle BDM = \angle CEM$$
$$\angle BMD = \angle CME.$$

The first of these three subgoals is a premise of our original subgoal. The second can be proved making use of the tactic: 'if you want to prove two angles equal prove they are both right angles', this last also being given in the premises of the original goal. The third subgoal can be proved making use of the tactic: 'if you want to prove two angles are equal, then prove that they are vertically opposite

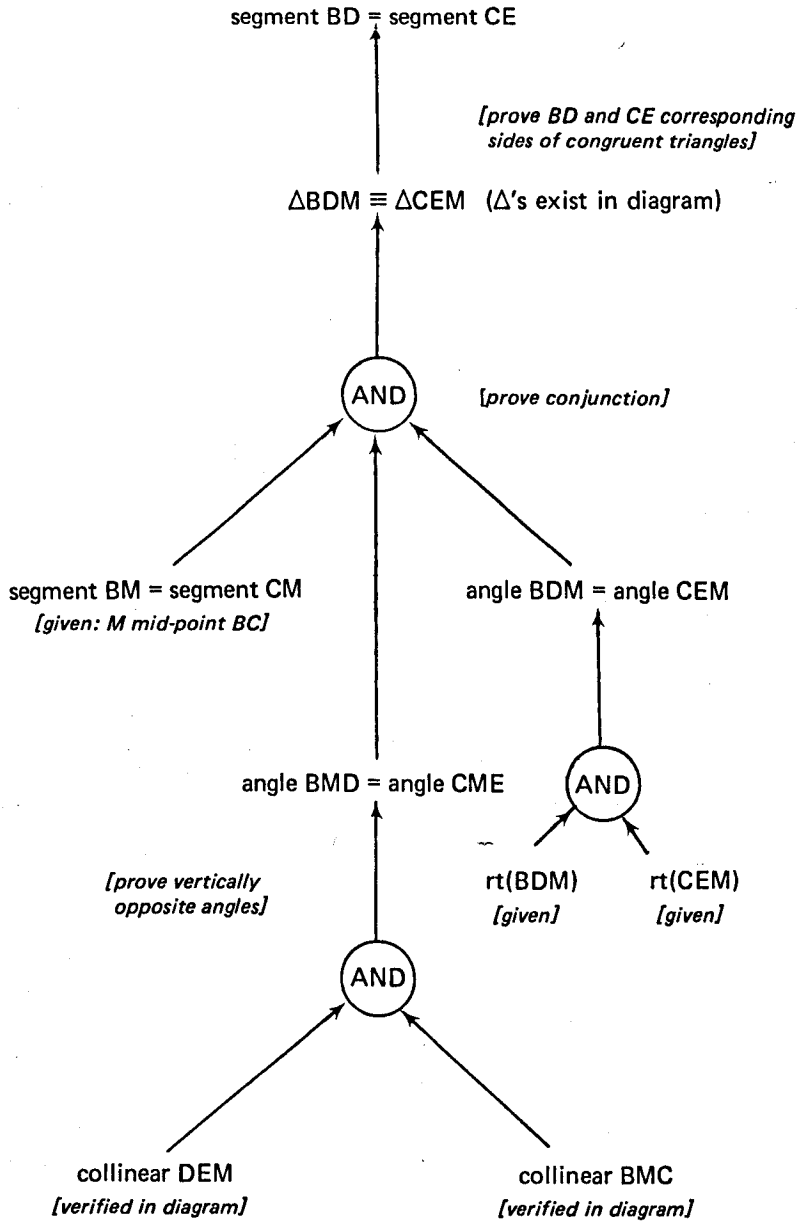


FIG. 2

angles'. This last involves descendent subgoals to establish the collinearity of D, M and E and the collinearity of B, M and C. As in some earlier examples, these subgoals can be "established" by procedures whose domain is the diagram. The final proof tree is shown in Figure 2.

Summarising: in this introduction we have attempted to both illustrate a

proof style and indicate the role of a diagram in facilitating proof discovery within that style.

The proof style essentially uses just one kind of tactic of the general form: 'if you want to prove B and you know a theorem 'if A_1 and A_2 and .... and A_n are true, then B is true' then try independently to prove A_1, A_2 ... and A_n". This proof style has been given the descriptive name 'backward chaining': as already seen, it can be illlustrated by a proof tree which is complete when all terminal nodes are 'givens' (or validated directly by procedures acting on the diagram)–i.e. when we have managed to chain backward from the theorem to be proved to these givens or things 'obviously true in the diagram'.

A *proof tree*, of which an example is shown in Figure 2 does not, of course, illustrate the full process of *proof search*. As we have attempted to indicate in the informal sketch above, the process of backward chaining might set up a subgoal for which a number of tactics might be applicable. Each of these, in turn, gives rise to a subtree in the search tree. Many of these a-priori applicable tactics might turn out, when examined, to be inappropriate. However, the discovery of their inappropriateness might involve elaboration of the subtree to some depth. The growth of the proof search tree is potentially explosive and it is vitally important that its growth be controlled and, in particular, subtrees which are going to fail (more precisely, cannot be part of a proof tree), should be detected and their exploration abandoned as soon as possible.

In the sketch above we have illustrated the role of the diagram as a factor in this control of the generation of (irrelevant) subtrees by rejection of proposed subgoals (root nodes of potentially large subtrees) which can be shown to be false in the diagram by computational procedures (as opposed to formal proof in the axiomatic system) over (the points of) the diagram.

This is not the only control mechanism which might be operative in the search process. For example, given a subgoal and given a set of potentially applicable strategies, it might be possible to order the alternative strategies according to some likelihood criterion perhaps based on some context in which the subgoal is embedded. This last kind of control mechanism has been less well explored and is less well understood. It will not be of concern in this paper.

In part 2 below we will examine briefly some work on the implementation of a geometry machine which follows the paradigm of part 1. As part of this, some of the points covered in part 1 will be made precise in a precise context. Weaknesses as well as strengths of current work in the paradigm will be considered and an attempt made to indicate how a 'seeing' machine geometer might develop.

PART 2

In two fascinating papers written fifteen years ago (Gelernter, 1959 and Gelernter, Hansen and Loveland, 1960), the authors wrote about what they called a geometry theorem proving machine. The 1960 paper begins with the (stirring) words:

> "In early spring, 1959, an IBM 704 computer, with the assistance of a program comprising some 20,000 individual instructions, proved its first theorem in elementary Euclidean plane geometry (Gelernter, 1959b). Since that time, the geometry-theorem proving machine (a particular state configuration of the IBM 704 specified by the aforementioned machine code) has found solutions to a large number of problems taken from high school textbooks and final examinations in plane geometry. Some of these problems would be considered quite difficult by the average high school student. In fact, it is doubtful whether any but the brightest students could have produced a solution for any of the latter group when granted the same amount of prior "training" afforded the geometry machine (i.e., the same vocabulary of geometric concepts and the same stock of previously proved theorems)."

The papers, whilst leaving much to be inferred by the reader, make clear that the 'geometry theorem proving machine' is based on the powerful paradigm described informally in part 1 of this paper. However, until very recently little attempt was made to build on this work. The ensuing years have seen an emphasis on the mechanization of complete uniform proof procedures for first order predicate calculus. It has become increasingly clear that this work by itself is unlikely to take one into the domain of interesting theorems. It now seems generally accepted that proof procedures must be capable of exploiting the specificity of the problem domain, be it geometry, number theory, whatever. Reiter (Reiter, 1972) discusses possible alternative ways of exploiting specificity and gives reasons for focussing attention on a particular extension of the paradigm of part 1 of this paper. We will try to indicate why later. First let us return to the first implementation by Gelernter and his co-workers. As mentioned earlier, their papers left much of their method to be inferred. In what follows and indeed in part 1 we have made use of Gilmore's careful and detailed analysis (Gilmore, 1970) to which readers are referred for a more formal treatment.

The geometry machine uses a given set of universally quantified statements (axioms) of the general form:

for all $x_1, x_2, \ldots x_n$:
if S_1 and S_2 and ... and S_n *then* S;

where $x_1, x_2 \ldots x_n$ are variables which are to be instantiated by (replaced consistently by) names of points. $S_1, S_2 \ldots S_n$ and S are applications of simple predicates of geometry such as:

triangle $(x_1 x_2 x_3)$;
collinear $(\{x_1 x_2 x_3\})$; between $(x_1 \{x_2 x_3\})$;
equal (segment $(x_1 x_2)$, segment $(x_3 x_4)$);
equal (angle $(x_1 x_2 x_3)$, angle $(x_4 x_5 x_6)$);

congruent (triangle $(x_1x_2x_3)$, triangle $(x_4x_5x_6)$);
mid-point $(x_1$, segment $(x_2x_3))$

etc. or their negations.

For example (leaving the statement of universal quantification over the variables as understood):

if between $(x_2\{x_1x_3\})$ and between $(x_2\{x_4x_5\})$
then equal (angle $(x_4x_2x_3)$, angle $(x_1x_2x_5)$);
if distinct $(\{x_1x_2x_3\})$ and not (collinear $(\{x_1x_2x_3\})$);
then triangle $(x_1x_2x_3)$.

Apart from simple substitution, the only other mechanism for deriving theorems is the simple (inference) rule:

Given the axioms

if S_{11} and S_{12} and . . .and S_{1n} *then* S_1;
if S_{21} and S_{22} and . . .and S_{2m} *then* S_2;
if S_1 and S_2 *then* S;

we can conclude

if S_{11} and . . . and S_{1n} and S_{21} and . . . S_{2m} *then* S.

The theorems of the system have precisely the same form as the axioms: they are all (universally quantified) implication sentences.

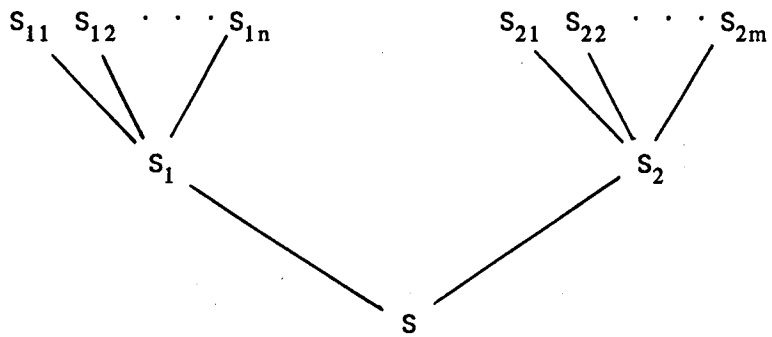


FIG. 3

The inference rule can be expressed by the tree of Figure 3. From this stems the notion of a proof of a theorem

if S_1 and S_2 . . . and S_k *then* S

as a tree in which nodes are labelled with sentences and:

(i) each node is labelled with a simple sentence in the set $S_1 \ldots S_k$, or

(ii) is connected to a set of descendant nodes labelled with the antecedent sentences of some axiom with consequent the label S of the parent node.

An example of such a proof tree has been given in Figure 2.

The theorem proving algorithm of the geometry machine is, as already intimated, based on 'backward chaining' which can now be expressed more formally as a process for searching for a proof tree by starting from a root node (labelled with the consequent of the theorem to be proved) and exploring the *set* of trees which can be generated at any stage by the inference rule and the set of applicable axioms (those with consequent sentences labelling a terminal node of the tree).

Gilmore (Gilmore, 1970) shows that this process is both a theorem proving algorithm and a decision process. By a decision process is meant that if an implication sentence is (is not) a theorem in the particular system defined by the particular set of implication sentences taken as axioms, then the process will terminate successfully (unsuccessfully). Being a theorem proving algorithm implies that successful termination also returns the proof tree.

The particular axiom set used in the Geometry Machine is not important here (other than to recognise, of course, that it determines the particular fragment of geometry captured by the machine) and we shall focus our attention on the formal counterpart in the theorem proving algorithm of the Geometry Machine of the paradigm use of a diagram in the proof style of part 1.

In order to do this with some precision, we need to explain the notion of a model. For this we return to the opening quotation from Kac & Ulam. Plane geometry is a first order axiomatic theory. It deals with undefined objects called points and lines and the system is defined by a (small) set of axioms stating relations which hold over the objects of the system together with a method of inference (that of first order predicate calculus) which allows new relations to be deduced–the new relations being called theorems. The proof of a theorem in the system consists in exposing its generating chain of inferences: so-called syntactic proof.

Alternatively it is possible to set up a mechanism for assigning a meaning to a well-formed sentence in the system. This is done by choosing some definite domain D of objects and mapping the objects, function and predicates of the well-formed sentence in the system onto objects in D and functions and relations over D respectively. Such a mapping is called an interpretation or model of the well-formed sentence and the sentence will have a truth value in this model. The notion of a theorem in the axiomatic system now becomes that of a well-formed sentence which is true in all models: a so-called semantic notion of proof. It turns out that the syntactic and semantic notions of proof are equivalent: i.e. categorize the same set of sentences. The second notion, however, has an interesting property. Since a sentence is a theorem if and only if it is true in *all* models, disproof can simply consist in exhibiting a *single* model in which the

sentence is false (the method of counter example). It is this last which lies at the heart of the use of diagrams in the Geometry Machine.

The models we shall use in the Geometry Machine will be ones in which D is the domain of ordered real number pairs. A named point in a theorem to be proved will be mapped into a particular pair of D (conventionally: its coordinates in the Cartesian plane). A line determined by two points P_1P_2 is mapped into the set of pairs (x,y) defined by the *algebraic* relation

$$y\text{-}y_1/x\text{-}x_1 = y_2\text{-}y_1/x_2\text{-}x_1.$$

Other geometrical functions and predicates are mapped into their usual algebraic interpretations in the Cartesian plane. We can now show that a sentence is not true by simply showing that it has a denotation in the Cartesian plane which is false.

How does this help us? First, let us clarify the relationship between a theorem and a diagram. A theorem refers to a set of named points and to certain relationships holding over them. The function of a diagram is to explicate in some model the denotations of these particular relations out of the total set of relations holding over the set of points.

In the particular case of an implication sentence

"if S_1 and S_2 . . . and S_n then S"

for the Geometry Machine, the diagram would consist of a set of number pairs, one for each point named in the implication sentence, and chosen so that the premises S_1 to S_n of the sentence are true in the diagram. The general properties of the (Cartesian) model guarantee that the axioms of the Geometry Machine are true in the model. It follows that anyting false in the model is not derivable from the axioms and the premises of the implication sentence to be proved.

Again, how does this help us? It gives us the possibility (illustrated informally in part 1) of mediating the search for a syntactic proof by semantic notions. For example, it might be desirable to establish at a particular point in proof search whether a relation such as "mid-point $(P, \{P_1P_2\})$" holds or not. Computationally it might be difficult or just lengthy to decide this by syntactic methods. On the other hand, if (x,y), (x_1y_1) and (x_2,x_2) are the number pairs in the diagram denoted by P, P_1 and P_2 respectively, then a simple arithmetic evaluation of the expressions

$$2x\text{-}x_1\text{-}x_2$$
$$2y\text{-}y_1\text{-}y_2$$

resulting in a value for either which is sensibly different from zero makes it obvious that the relation is false in the diagram and, therefore, not derivable syntactically. On the other hand, if both these expressions are close to zero (machine arithmetic with finite precision!), then although this cannot be taken as establishing the relation, it might be taken as an indication that the effort of

examining the truth of the relation by syntactic methods was worthwhile.

Examples of arithmetic evaluation in the diagram abound: they parallel the 'perceptual computations' on the ink-mark drawings which were used as diagrams in part 1. We clearly have considerable potential here for a rich interplay of syntax and semantics in proof search.

Not all these possibilities are exploited in the Geometry Machine: we will finish this part by a fairly abstract characterization of the particular use the Geometry Machine, as described so far, makes of the diagram. In part 3 below we shall briefly explore other possibilities.

The Geometry Machine is given:

(i) an implication sentence
if S_1 and S_2 and . . . and S_n then S
to prove;

(ii) a denotation of each named point in $S_1 \ldots S_n$ as a number pair, the number pairs being carefully selected to make each S_i $1 \leqslant i \leqslant n$ true as discussed above;

(iii) the set of mentioned line segments in S_i $1 \leqslant i \leqslant n$ and S where constructs, such as triangle (ABC), formed from line segments, are treated as a mention of the implied line segments.

In searching for a proof tree, the Geometry Machine will only use simple sentences that are true in the diagram (where "truth in the diagram" has the meaning already discussed).

Since the Geometry Machine has only a finite number of functions and predicates, and since the implication sentence to be proved refers to only a finite number of points, and since the inference rules of the Geometry Machine do not allow new named points to be generated: there are just a finite number of simple sentences which are true in the diagram and these could be computed once and for all. If we call this set of simple sentences D_T then we can assert that if the implication sentence

if S_1 and S_2 and . . . and S_n *then* S

is a theorem, then S must be a member of D_T. The diagram can be regarded simply as a convenient device for computing the set D_T. The set D_T in general will be vastly smaller (for the reasons explored above) than the total set of simple sentences that can occur in the set of *all* (syntactically allowable) substitution instances of axioms using the point names of $S_1 \ldots S$, D_I say.

In searching for a proof tree, we need only use sentences from D_T to label nodes. The vast difference in size between D_T and D_I is another way of characterizing the exploitation of the diagram in the Geometry Machine theorem proving algorithm.

Another use of the diagram by the Geometry Machine parallels important

operations mentioned in part 1 which informally made use of certain kinds of relations among points which could be said to be "obvious from the diagram" and not requiring a (sometimes irritatingly tedious) syntactic proof. Terminal nodes in the proof tree are allowed to be labelled by certain sentences in D_T which are not premises but which would stem typically from axioms of order or from axioms necessary because "although the geometry is a point geometry, some of the simple sentences of the axioms are expressed in terms of angles, line segments. . ." (Gilmore, 1970). For example, if the diagram contains the equivalent of Figure 4 then a terminal node of a putative proof tree might be labelled:

angle (ABD) = angle (EBC)
[vertically opposite angles in diagram].

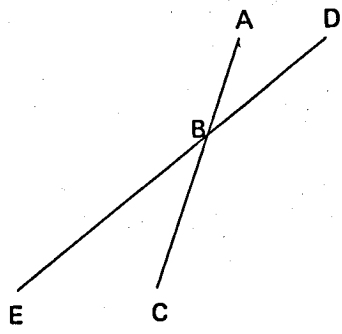


FIG. 4

The justifying axiom (not made overt in the system) would be: "*if* A B C are collinear in that order and D B E are collinear in that order, *then* angle (ABD) = angle (EBC)". The organization of the computational procedures that embody such axioms and the method by which they are invoked in the actual Geometry Machine is not clearly stated. Nevertheless, this judicious blurring of the notion of syntactic proof and truth in an appropriate model for certain kinds of sentences is one that lies at the heart of doing mathematics, and which Reiter (Reiter, 1972) has shown must be a component of a good theorem-prover.

Finally, the Geometry Machine also uses the set of given line segments to order the substitution instances considered. If an axiom mentions a line segment, then only line segments from the initially given set are used as substitution instances until all possible such substitutions are exhausted without proof. If this happens, then arbitrary point pairs in the given list for the diagram are used to define new line segments for substitution. These new line segments are added to the diagram list and could be regarded as a form of "weak" construction. The reason for ordering is obvious, but not overly convincing as a motivating mechanism for controlling search.

It should be emphasised that these weak constructions do not introduce any

new named points, only line segments between existing named points. The axioms of the Geometry Machine are such that all named points in the conclusion of an implication sentence are mentioned in at least one of its antecedents: no mechanism exists in the Geometry Machine for the introduction of new points. However, without such a mechanism the Machine is cut off from the more interesting class of theorems. In an attempt to remedy this deficiency, the Geometry Machine was, indeed, extended by the addition of a single axiom involving existential quantification. The axiom asserts that "if a line segment xy is not parallel to a line segment zw then there exists a point of intersection u (xy∩zw)": i.e.

for all xyzw

if not (parallel (xy,zw))

then there exits u *such that*

collinear (x y u) and collinear (z w u).

However, this axiom is used quite differently to those introduced earlier. In particular, it is never used to label a node in a putative proof tree: rather it is invoked only when the current search space is exhausted. The axiom, if applicable, is then used to introduce a new named point into the diagram and the process of search for a proof tree started again. It is clear that the notion of proof is unaltered in the extended machine, but the theorem proving algorithm is now no longer a decision process as well since the existential axiom allows for non-terminating successive introduction of new points.

Since constructions produced by this new axiom are so ill-motivated, and since we will discuss constructions in a wider context in part 3 below, the point will not be elaborated here. Rather, we will let the undoubted merit of the seminal work on the Geometry Machine be judged from the described unextended machine.

PART 3

As mentioned earlier, it is only comparatively recently that interest in the Geometry Machine has revived.

One factor has been the development of very high-level goal oriented languages such as Planner (Hewitt, 1971) which are claimed to provide a powerful and natural formalism for structuring mathematical knowledge as programs. Certainly, language constructs in Planner are well-matched to the kind of tactical interpretation of axioms and backward chaining of the Geometry Machine. Indeed, a Geometry Machine in the style of the original was implemented very straightforwardly by Goldstein (Goldstein, 1973) using just such a language. The description of this implementation is detailed enough to indicate the total set of implication sentences used to achieve the level of performance indicated.

Goldstein also discussed the possibility of extending the Geometry Machine

to obtain a more motivated search for a proof and sketches how this might be coupled with "knowledge" for making constructions in the diagram. This last is not presented as a uniform procedure, but rather as if controlled by a set of heuristics associated with particular goals or strategies. This in a sense prejudges the needs of the set of infinite individual proofs. However, it is difficult to infer enough from the few remarks made to be quite sure what is intended. Since this point of constructions being a response to the evolving total state of a syntactic proof is a central issue in some recent careful and detailed work by Reiter (Reiter, 1972), we will base our discussion on his work.

This discussion will return to the informal style of part 1 and with the same intent: simply to give a feeling for the problems and possibilities of a major extension of the proof methods of the Geometry Machine. Readers interested in a fuller and inevitable formal discussion are referred to Reiter's paper in which the complete inference system is given and motivated by some nicely designed examples which illustrate its potential power.

As far as the author is aware, there is as yet no machine implementation based on these ideas and, indeed, it is clear that there are challenging problems to be solved in a design for an implementation.

The idea to be explored briefly below is a continuation of the theme of the desirability of a rich interplay between syntax and semantics and particularly that aspect concerned with the generation of constructions appropriate to the evolving syntactic proof. The idea will be explored in a very informal treatment of an example taken from Reiter's paper.

Suppose then we want to prove the following theorem (implication sentence):

> *If* ABCD is a trapezoid *and* BC is parallel to AD *and* the line joining the mid-point E of AC to the mid-point F of BD meets AB at M *then* MA = MB.

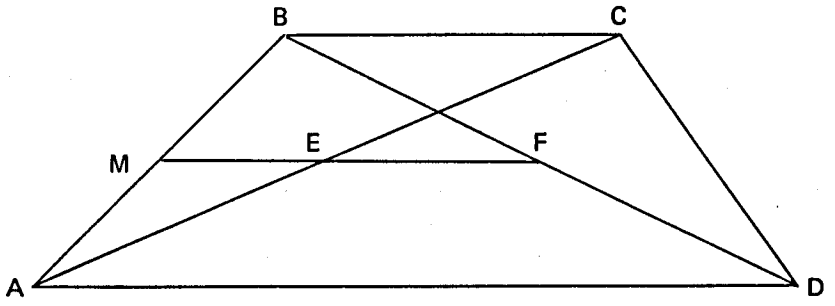


FIG. 5a

Figure 5(a) shows a drawing of the intial diagram (model). We notice that MF is parallel to AD in the model: i.e., say

$(y_D\text{-}y_A)(x_F\text{-}x_M) \cong (y_F\text{-}y_M)(x_D\text{-}x_A)$.

If we can, indeed, prove this parallelism syntactically, then an application of the axiom

if Δxyz and collinear (xuy) and collinear (xvz) and parallel (uv,yz) and xv = vz
then xu = uy

with the substitution instance ΔBAD with BF = FD given, and parallel (MF,AD), will allow BM = MA (our original goal) to be inferred.

The problem then is to prove parallel (MF,AD).

We are here touching on the extremely interesting topic of attending to (relevant) truths in the model. Here we must simply leave as an open question whether exploration of truth in the model should always be in response to goals in a developing syntactic proof or whether some prior exploration of truth in the model might suggest appropriate sets of syntactic possibilities and facilitate their ordering for detailed consideration.

To return to the problem of producing a syntactic proof of parallel (MF,AD). Computation in the model produces predicates parallel (ME,AD) and parallel (EF,AD) as equivalent to parallel (MF,AD): their equivalence can be justified syntactically by the axiom

A1: if parallel (xy,uw) and collinear (uvw)
then parallel (xy,uv).

This axiom can be interpreted as making a remark about the model which bears on the possibility of just this issue raised earlier of syntactically motivated constructions. The remark is that, if you want to prove some instantiation (in the model) of

parallel (xy,uv)

then check by a computation in the model (on, say, Cartesian point coordinates) that the instantiation is true in the model, when you can also be sure that a consistent instantiation of

parallel (xy,uw) and collinear (uvw)

can be found which will be true in the existing model if there is a named point on uv in the model, or true in the existing model extended by introducing a new named point anywhere on uv to serve as an appropriate denotation of w. This extension would be a syntactic proof motivated construction in the model producing a new model in which the semantic requirements for the use of the axiom are met.

If the axiom A1 is, indeed, applied in this spirit to the goal parallel (EF,AD), we have the instantiation x/E, y/F, u/A, v/D. There is no named point in the

model which can instantiate w. We know however that there does exist an extension of the model in which this instantiation can be made and so we proceed to the attempt to prove

parallel (EF,Aw) *and* collinear (AwD).

In the spirit of deferring this particular extension of the model (instantiation of w) we try first to prove

parallel (EF,Aw).

Assume the axiom:

A2: *if* Δxyz *and* collinear (xry) *and* collinear (xsz)
and xr=ry *and* xs=sz
then parallel (rs,yz).

An appropriate instantiation is r/E, s/F, y/A, z/w and again the use of the axiom can be justified semantically only if the instantiation can be completed so that

ΔxAw and collinear (xEA) and collinear (xFw)
and xE = EA and xF = Fw

is true in some extension of the model in which w is instantiated by a point on AD (this last constraint on extension making it unnecessary to return and check the other predicate of the conjunction above). Now the predicates

collinear (xEA) *and* xE = EA

give x a unique instantiation, C, in the model and it is also quite easy to conceive of appropriate computations (say over a "list-of-points-on" EA initiated by the collinear predicate etc.) which would find this instantiation. Now, given the instantiation x/C we have to satisfy

collinear (xFw) *and* Fx = Fw

i.e.

collinear (CFw) *and* CF = Fw

in the model. This time there is no instantiation of w in the model. The predicates, however, fix the denotation of w in an extended model uniquely. However, an extended model has to satisfy collinear (AwD). The instantiation of w in the extended model has, therefore, to satisfy

ΔCAw *and* collinear (CFw) *and* FC = Fw *and* collinear (AwD).

The point K (CF∩AD) in Figure 5(b) is readily computed as the required unique instantiation of K which makes the conjunction of assertions true in the (extended) model.

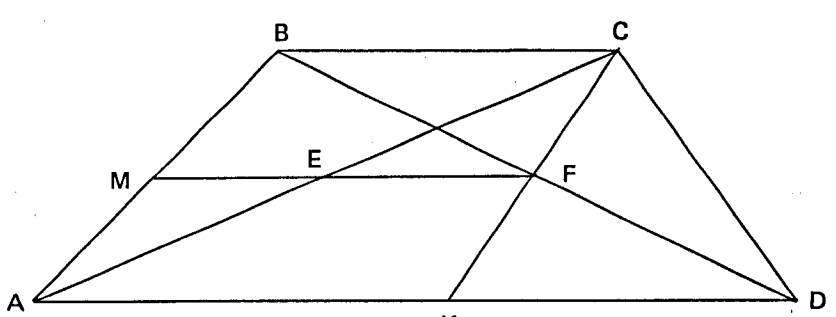


FIG. 5b

Taking K then to be the point named by CF∩AD, we continue the syntactic proof and try to establish

$$\Delta CAK \textit{ and } \text{collinear (CEA) } \textit{and } \text{collinear CFK}$$
$$\textit{and } \text{collinear (AKD) } \textit{and } EC = EA \textit{ and } FC = FK.$$

The first five literals of this conjunction are easily proved leaving the goal

$$FC = FK$$

still to be established. This can be done by proving the Δ's FKD and FCB congruent.

The example has been carried far enough for our expository purpose, that is, to illustrate the possibilities of interaction of syntactic and semantic methods in proof search, and to make it plausible that the gap between our informal presentation of an extended proof style and an extended machine is worthy of serious research and a continuance of an excellent tradition.

REFERENCES

Gelernter, H. (1959) Realization of a geometry theorem proving machine. *Proc. Int. Conf. Inform. Processing*, Paris.

Gelernter, H., Hansen, J.R., and Loveland, D.W. (1960) Empirical explorations of the geometry theorem machine. *Proc. Western Joint Computer Conf.*

Gilmore, P.C. (1970) An examination of the geometry machine. *Artificial Intelligence*, 1, 171-187.

Goldstein, I. (1973) Elementary geometry theorem proving. *A.I. memo. 280*, A.I. Laboratory, M.I.T., Cambridge, Mass.

Hewitt, C. (1971) Procedural embedding of knowledge in PLANNER. *Proc. IJCAI-2*, London. Portsmouth: Eyre and Spottiswoode Ltd.

Kac, M. and Ulam, S. (1971) *Mathematics and Logic*. Penguin Books, England.

Reiter, R. (1972) The use of models in automatic theorem proving. *Tech. Report 72-09*, Dept. of Computer Science, Univ. of British Columbia.

2

Typing and Proof by Cases in Program Verification

W. W. Bledsoe and Mabry Tyson

Departments of Mathematics & Computer Science
University of Texas at Austin

Special procedures have been added to an automatic prover to facilitate its handling of inequalities and proof by cases. A data base, called TYPELIST, is used which maintains upper and lower bounds of variables occuring in the proof of a theorem. These procedures have been coded and used to (interactively) prove several theorems arising in automatic program verification.

INTRODUCTION

We describe here procedures that have been added to an automatic theorem prover (Bledsoe and Tyson, 1975) to make it more effective in proving verification conditions (theorems) that arise in the field of program verification. These procedures, which handle inequalities and equalities, and proof by cases, are based on a pointer system used by Bundy (Bundy, 1974), SRI (Rulifson, *et al*, 1972; Waldinger and Levitt, 1974), and others to handle inequalities, and on the interval types used in (Bledsoe, *et al*, 1972). The present description follows somewhat the discussion in (Bledsoe, 1974).

In order to follow this presentation the reader should have some understanding of the prover described in (Bledsoe and Tyson, 1975). However, we feel that many workers in this field are already familiar with our prover and can read this paper directly, referring to the previous paper only when the need arises. Tables I and II from that paper are included here as Appendix A, for convenience, but the reader is referred to Section 2 of (Bledsoe and Tyson, 1975) for a fuller understanding.

These methods can also be used in Resolution-based provers and other Gentzen type systems ('Typelist for resolution', below).

TYPES

Typing information can be a powerful asset in automatic theorem proving. For example, knowing that j and k are non-negative integers and that $j < k$ lets us deduce that $j \times k \geq 0$, $j \leq k\text{-}1$, etc. Often, we have other "typing" infor-

mation. For example, we may know (from a given hypothesis) that j lies in some interval, $a \leqslant j \leqslant b$. In our system we have decided to include such information as part of the type of j. Thus j has the type: "non-negative integer in the interval $a \leqslant j \leqslant b$". We express this fact by the notation $\{j: a\ b\}$.

In what follows, certain variables i, j, k, . . . occur in inequalities and can assume only non-negative integer values. These will be "typed" as indicated above. Such variables often arise as program variables in computer programs. (Actually these variables are all universally quantified in the theorem being proved and are converted to skolem constants by the skolemization process, but that need not concern us here. Refer to Appendix 1 and Section 1 of (Bledsoe and Tyson, 1975).)

Upper and lower bounds are computed and maintained for these typed variables. When a new inequality is encountered, as a hypothesis, the bounds for these variables are updated appropriately. This interval information is kept in a knowledge base (which we call the TYPELIST), which represents the "state of the world" for these variables at that particular time, and serves as an additional hypothesis to the theorem or subgoal being considered. For example, a hypothesis

$$(a \leqslant j \leqslant b)$$

is stored in TYPELIST as

$$\{j: a\ b\}$$

which means that j is in the closed interval [a,b].[1] If a contradiction such as $\{j: k\ k\text{-}1\}$ occurs in TYPELIST, this represents a false hypothesis and successfully terminates the proof. Also, if an entry $\{j: N\ \infty\}$ is already in TYPELIST, any new hypothesis such as $(j \leqslant N+1)$ causes the entry to be updated to $\{j: N\ N+1\}$, which means that j can take only the value N or the value N+1.

An entry of the form $\{j: N+1\ N+1\}$ which occurs in TYPELIST is treated as the equality $(j = N+1)$.

Initially all typed variables j are given the type $\{j: 0\ \infty\}$.

A subroutine SET-TYPE is used to convert information in the hypothesis of a theorem to TYPELIST entries. It is called at the beginning of the proof and at each point in the proof when new expressions are added to the hypothesis of the theorem being proved. For example, if the theorem being proved is

Ex. 1.

(1) $\quad (P(1) \wedge 1 \leqslant j \wedge j \leqslant n \wedge j \leqslant 1 \rightarrow P(j))$

the original value of TYPELIST is

$$(\{j: 0\ \infty\}\ \{n: 0\ \infty\}),$$

[1] Except in the case where b is $+\infty$; the interval is then [a, "].

but then SET-TYPE is called on the hypothesis of (1) which changes TYPELIST to

$$(\{j: 1\ 1\}\{n: j\ \infty\})$$

and converts (1) to

(2) $(j=1 \wedge P(1) \rightarrow P(j))$.

Notice that the program detected that j was equal to 1 from the entry $\{j: 1\ 1\}$. The prover will now substitute 1 for j in (2) to obtain

$$(P(1) \rightarrow P(1))$$

which it recognizes as true.

Other examples are now given.

Ex. 2.

(3) $(1 \leqslant j \wedge P(1) \rightarrow (j \leqslant k \wedge k \leqslant 1 \rightarrow P(k)))$.

An initial call to SET-TYPE, on the hypothesis of (3), changes TYPELIST to $(\{j: 1\ \infty\}\{k: 0\ \infty\})$ and converts (3) to

(4) $(P(1) \rightarrow (j \leqslant k \wedge k \leqslant 1 \rightarrow P(k)))$.

Now Rule 7 of IMPLY (Bledsoe and Tyson, 1975, Table I), converts (4) to

(5) $(P(1) \wedge j \leqslant k \wedge k \leqslant 1 \rightarrow P(k))$

at which time SET-TYPE is again called, which uses $j \leqslant k$ and $k \leqslant 1$ to change TYPELIST to $(\{j: 1\ 1\}\{k: 1\ 1\})$, and converts (5) to

$$(j=1 \wedge k=1 \wedge P(1) \rightarrow P(k)).$$

The prover, as before, converts this to

$$(P(1) \rightarrow P(1))$$

which it recognizes as true.

Ex. 3.

$$(2 \leqslant j \wedge j \leqslant 1 \rightarrow P(j))$$

SET-TYPE changes TYPELIST to $(\{j: 2\ 1\})$. The program detects the contradictions in TYPELIST (i.e., $2 \leqslant 1$) and successfully concludes the proof.

Whenever an inequality $(a \leqslant b)$ occurs in the conclusion of the theorem being

proved, the prover updates TYPELIST with the negation of $(a \leqslant b)$, and looks for a contradiction. Thus, for the example

Ex. 4.

(6) $(j \leqslant 1 \wedge k \leqslant j \wedge P \rightarrow k \leqslant 3)$,

TYPELIST is given the value $(\{j: k\ 1\}\ \{k: 0\ j\})$ and (6) is converted to

$$(P \rightarrow k \leqslant 3).$$

The prover now uses $(k \not\leqslant 3)$, which is first converted to $(4 \leqslant k)$[2] to update TYPELIST, getting $(\{j: k\ 1\}\ \{k: 4\ j\})$, which contains the contradiction

$$(4 \leqslant k \leqslant j \leqslant 1).$$

The prover detects such contradictions by computing absolute upper and lower bounds, sup and inf, for j and k. For this case

$$\sup j = 1, \quad \inf j = 4$$
$$\sup k = 1, \quad \inf k = 4.$$

Since $4 > 1$ we have a contradiction. The prover uses the routines SUP and INF to evaluate these bounds. In (Bledsoe, 1975) we carefully define the algorithms SUP and INF and prove that they have the required properties.

Formula (6), (without the P), is an example of a formula in Presburger Arithmetic. These often arise from computer programs, and are discussed in (Bledsoe, 1975) and by Cooper (Cooper, 1971).

Ex. 5.

$$(2 \leqslant j \leqslant 4 \wedge k \leqslant j \wedge k \leqslant 7 \rightarrow C)$$

Here we use the symbols 'max' and 'min' in typing j and k. TYPELIST is given the value $[\{j: \max(2,k)\ 4\}\ \{k: 0\ \min(j,7)\}]$.

TYPELIST IN PROVER

In Section 2 of (Bledsoe and Tyson, 1975) we describe IMPLY and HOA, the main algorithms of Prover, and give Tables I and II which define them, and list several examples of their use. (Tables I and II are reproduced as Appendix A of this paper for convenience.) The reader is referred to the previous paper for a more detailed understanding.

IMPLY has five arguments

(TYPELIST, H, C, TL, LT)

---

[2] Since k is an integer (Bledsoe and Tyson, 1975, p. 27).

but in Section 2 of (Bledsoe and Tyson, 1975) we deal only with H, C, and TL, the hypothesis, conclusion, and theorem label of the theorem or subgoal being proved. For convenience to the reader we represent, in this paper, a call to IMPLY(TYPELIST, H, C, TL, LT) by the notation

(TL) $(H \Rightarrow C)$.

As mentioned earlier, TYPELIST represents an additional hypothesis, so we will augment this notation as follows:

(TL) $([TYPELIST] \wedge H \Rightarrow C)$.

Thus after Example 2 is partially converted it is represented by

(1) $([\{j: 1\ 1\}\{k: 1\ 1\}] \wedge P(1) \Rightarrow P(k))$.

We will now describe some changes and additions to the Rules of IMPLY and HOA (in the form of amendments and additions to Tables I and II, Appendix A) which have been made to facilitate the use of TYPELIST. Before doing so we first describe the algorithm SET-TYPE, which was mentioned earlier.

SET-TYPE(A)

This algorithm updates TYPELIST by using inequalities and equalities in conjunctive positions of A, and returns a value A', which is the remainder of A not used in updating TYPELIST.

For example, if TYPELIST = $[\{j: 0\ k\}\{k: j\ 7\}]$ then a call

SET-TYPE($k \leqslant 5 \wedge P(j)$)

updates TYPELIST to

$[\{j: 0\ k\}\{k: j\ 5\}]$

and returns the value P(j).

IMPLY rule changes

The definition of IMPLY given in (Bledsoe and Tyson, 1975, Table I) and in Appendix A is changed by replacing Rule 7 and adding new Rules 11 and 14 as given below:

RULE	IF	ACTION	RETURN
7.	C ≡ (A→B)	Put A' := SET-TYPE(A) Let TY' be the updated version of TYPELIST	
7.1	TY' has a contradiction		"T"
7.2	ELSE		IMPLY(TY', H ∧ A', B)

(table continued)

11.	C ≡ (a<b)	Put A' := SET-TYPE(~(a<b)) Let TY' be the updated version of TYPELIST	
11.1	TY' has a contradiction		"T"
11.2	TY' ≡ TYPELIST	Go to 12 (with C and TYPELIST as they were)	
11.3	TY' ≠ TYPELIST		(T TY')
14.1	C ≡ (a=b)	Put C' ≡ (a<b ∧ b<a)	IMPLY(H, C')
14.2	C ≡ (a≠b)	Put C' ≡ (a<b ∨ b<a)	IMPLY(H, C')

Later in this description we will further change these tables, but the reader need not be concerned with that at this time. We will summarize all of these changes in Tables I-T and II-T.

Ex. 5.

$$(Q \rightarrow (j \leqslant 1 \wedge k \leqslant j \wedge P \rightarrow k \leqslant 3))$$

(1) $([\{j: 0\ \infty\}\{k: 0\ \infty\}]$
$\Rightarrow (Q \rightarrow (j \leqslant 1 \wedge k \leqslant j \wedge P \rightarrow k \leqslant 3)))$

Note that each of j and k is given the original type [0 ∞), when the theorem is given to the prover.

(1) $([\{j: 0\ \infty\}\{k: 0\ \infty\}] \wedge Q$

$\Rightarrow (j \leqslant 1 \wedge k \leqslant j \wedge P \rightarrow k \leqslant 3))$ I7

In this case SET-TYPE(Q) left TYPELIST unchanged and returned the value Q.

$$\text{(1)}\quad \overbrace{([\{j: k\ 1\}\{k: 0\ j\}]}^{\text{TYPELIST}} \wedge \overbrace{(Q \wedge P)}^{\text{H}} \rightarrow \overbrace{k \leqslant 3)}^{\text{C}} \qquad \text{I7}$$

Here SET-TYPE(j≤1 ∧ k≤j ∧ P) has updated TYPELIST to the new value shown, and returned P, which was conjoined to Q.

Now the new Rule I-11 employs SET-TYPE(~(k≤3)) = SET-TYPE(4≤k) to update TYPELIST to TY' = [{j: k 1}{k: 4 j}], and Rule 11.1 detects the contradiction

$$4 \leqslant j \leqslant 1$$

in TY' and terminates the proof successfully.

As mentioned above we detect the contradiction in

$$TY' = [\{j: k\ 1\}\ \{k: 4\ j\}]$$

(or any other list of inequalities) by computing

$$\sup_{TY'}(j) \text{ and } \inf_{TY'}(j).$$

In this case

$$\sup_{TY'}(j)=1,\ \inf_{TY'}(j)=4$$

and since $4>1$ we have a contradiction. These are computed by the algorithms SUP and INF [see (Bledsoe, 1975), especially Section 3]. In this example the values of sup and inf are rather obvious; for more involved examples, see Section 5 of (Bledsoe, 1975).

We have decided to give each variable j just *one* interval $\{j: a\ b\}$ in TYPELIST. So if we are proving a goal of the form

$$((j\leqslant 1 \vee j\geqslant 5) \wedge H \Rightarrow C)$$

where there is a *disjunction* of inequalities in the hypothesis, then we use *two* TYPELISTs expressed in the form

$$(([\{j: 0\ 1\}\ \{k:\ \}\ \ldots] \vee [\{j: 5\ \infty\}\ \{k:\ \}\ \ldots]) \wedge H \Rightarrow C).$$

To handle such examples we add Rule 2 to IMPLY to split such goals into two subgoals.

RULE	IF	ACTION	RETURN
2.	TYPELIST ≡ TY' v TY''	Put θ := IMPLY(TY', H, C)	
2.1	$\theta \equiv$ NIL		NIL
2.2	$\theta \not\equiv$ NIL	Put λ := IMPLY(TY'', H, C)	
2.3	$\lambda \equiv$ NIL		NIL
2.4	$\lambda \not\equiv$ NIL		$\theta \cdot \lambda$

Ex. 7.

$$(k\leqslant 3 \rightarrow k\leqslant 1 \vee 2\leqslant k\leqslant 3)$$

(1) $(\{k: 0\ \infty\} \Rightarrow (k\leqslant 3 \rightarrow k\leqslant 1 \vee 2\leqslant k\leqslant 3))$

(1) $(\{k: 0\ 3\} \Rightarrow k\leqslant 1 \vee 2\leqslant k\leqslant 3)$ I7
$(\{k: 0\ 3\} \wedge \sim(2\leqslant k\leqslant 3) \Rightarrow k\leqslant 1)$ H4.2
$(\{k: 0\ 3\} \wedge (k\leqslant 1 \vee 4\leqslant k) \Rightarrow k\leqslant 1)$
$((\{k: 0\ 1\} \vee \{k: 4\ 3\}) \Rightarrow k\leqslant 1)$

(1 1) $(\{k: 0\ 1\} \Rightarrow k\leqslant 1)$ I2

Rule 11' uses $\sim(k\leqslant 1)$ to update TYPELIST to $\{k: 2\ 1\}$ and Rule 11.1 detects the contradiction.

(1 2) $(\{k: 4\ 3\} \Rightarrow k \leqslant 1)$

Proved since $\{k: 4\ 3\}$ is a contradiction.

CASES

Many of the theorems (verification conditions) from program validation require a proof by cases, in that the theorem must be proved separately for two different ranges of values for some variable. Example 7 is such a case, but there the proof was straightforward because the two cases,

$$k \leqslant 1 \text{ and } 2 \leqslant k \leqslant 3$$

were stated explicitly in the theorem.

On the other hand, consider the following equivalent form of Example 7.

Ex. 8.

$$((k \leqslant 3 \wedge (k \leqslant 1 \rightarrow C) \wedge (2 \leqslant k \leqslant 3 \rightarrow C)) \rightarrow C)$$

(1) $(\{k: 0\ 3\} \wedge (k \leqslant 1 \rightarrow C) \wedge (2 \leqslant k \leqslant 3 \rightarrow C) \Rightarrow C)$ I7

Backchaining (Rule H7) off of the hypothesis $(k \leqslant 1 \rightarrow C)$ we obtain the subgoal

(1 H) $(\{k: 0\ 3\} \wedge (k \leqslant 1 \rightarrow C) \wedge (2 \leqslant k \leqslant 3) \rightarrow C) \rightarrow k \leqslant 1)$

which is false. Similarly if we backchain off of the hypothesis $(2 \leqslant k \leqslant 3 \rightarrow C)$ we fail again.

If the prover could somehow be made to know that it should consider the two cases

$$k \leqslant 1 \text{ and } 2 \leqslant k \leqslant 3$$

as it did in Example 7, the proof would proceed routinely.

We could, of course, require that prover backchain off of both of these hypotheses and thereby set up the provable subgoal

$$(k \leqslant 1 \vee 2 \leqslant k \leqslant 3)$$

but such a rule is not only unnatural, it is combinatorially explosive. What's more, a similar problem arises in many other theorems, such as

Ex. 9.

$$\begin{aligned}&(1 \leqslant n)\\ &\wedge \forall\, m(2 \leqslant n \wedge 1 \leqslant m \wedge m \leqslant 1 \rightarrow A[m] \leqslant A[2])\\ &\wedge \forall\, k(k+1 \leqslant n \wedge 2 \leqslant k \rightarrow A[k] \leqslant A[k+1])\\ &\rightarrow \forall K(K+1 \leqslant n \wedge 1 \leqslant K \rightarrow A[K] \leqslant A[K+1])\end{aligned}$$

and Example 10 below, which are more complicated that Example 8 and which will not submit to such an attack.

The procedure we employ to prove Example 8 and all others like it forces the prover into a proof by cases in a natural way. This is effected by further changes and additions to Tables 1 and 2. These are shown (for the most part) in Tables I-T and II-T below. These changes are justified by the results in Appendix II of (Bledsoe and Tyson, 1975A).

These changes require that IMPLY and HOA now return a pair

$$(\theta \ TY')$$

where θ is the same substitution we got before, and TY′ is a new value of TYPELIST which can be used in subsequent calls to IMPLY. This output value TY′ represents the part of the theorem that has not been proved. Thus if (θ TY′) is returned from a call IMPLY(TYPELIST,H,C), it means that (TYPELIST ∧ H→C) is valid except for the case TY′, or that

$$(\sim TY' \wedge TYPELIST \wedge H \rightarrow C)$$

is valid.

RULE	IF	ACTION	RETURN
2.	TYPELIST≡(TY′ v TY″)	Put Z := IMPLY(TY′, H, C)	
2.1	Z≡NIL		NIL
2.2	Z≡(θ TY1)	Put Z2 := IMPLY(TY″, H, C)	
2.3	Z2≡NIL		NIL
2.4	Z2≡(θ2 TY2)		($\theta \cdot \theta$2 (TY1 v TY2))
3.	H≡(A v B)	Put Z := IMPLY(TYPELIST, A, C)	
3.1	Z≡NIL		NIL
3.2	Z≡(θ TY1)	Put Z2 := IMPLY(TYPELIST, Bθ, C)	
3.3	Z2≡NIL		NIL
3.4	Z2≡(θ2 TY2)		($\theta \cdot \theta$2 (TY1 v TY2))
4.	C≡(A ∧ B)	Put Z := IMPLY(TYPELIST, H, A)	
4.1	Z≡NIL		NIL
4.2	Z≡(θ TY1)	Put Z2 := IMPLY(TYPELIST, H, Bθ)	
4.3	Z2≡NIL		NIL
4.4	Z2≡(θ2 TY2)		($\theta \cdot \theta$2 (TY1 v TY2))
7.	C≡(A→B)	Put A′ := SET-TYPE(A) Let TY′ be the updated TYPELIST	
7.1	TY′ has a contradiction		(T NIL)
7.2	ELSE		IMPLY(TY′, H ∧ A′, B)
9.	C≡(A = B)	IMPLY(TYPELIST, H, A ≤ B ∧ B ≤ A)	

(table continued)

11.	C ≡ (a < b)	Put A' := SET-TYPE(~(a < b)) Let TY' be the updated TYPELIST	
11.1	TY' has a contradiction		(T NIL)
11.2	TY' ≡ TYPELIST	Go to 12	
11.3	TY' ≠ TYPELIST[3]		(T TY')

TABLE I-T. IMPLY rule changes - TYPELIST version. IMPLY has arguments (TYPELIST, H, C, TL, LT), where H is the hypothesis and C the conclusion. We are ignoring TL and LT here.

[3] If TY' has an equality entry of the form {k: t t} then k is replaced by t in H, C, and TY'.

RULE	IF	ACTION	RETURN
4.	C ≡ A ∨ D	Put Z := HOA(B ~D, A)	
4.1	Z ≡ NIL		HOA(B ∧ ~A, D)
4.2	Z ≡ (θ TY1)	Go to 4.3	
4.3	TY1 ≡ NIL		(θ NIL)
4.4	TY1 ≠ NIL	Put Z2 := IMPLY(TY1, B∧~A, D)	
4.5	Z2 ≡ NIL		(θ TY1)
4.6	Z2 ≡ (θ2 TY2)		(θ •θ2 TY2)
6.	B ≡ A ∧ D	Put Z := HOA(A, C)	
6.1	Z ≡ NIL		HOA(D, C)
6.2	Z ≡ (θ TY1)	Go to 6.3	
6.3	TY1 ≡ NIL		(θ NIL)
6.4	TY1 ≠ NIL	Put Z2 := IMPLY(TY1, D, Cθ)	
6.5	Z2 ≡ NIL		(θ TY1)[4]
6.6	Z2 ≡ (θ2 TY2)		(θ •θ2 TY2)
7.	B ≡ (A→D)	Put θ := ANDS(D, C)	
7.1	θ ≡ NIL	Go to 7E	
7.2	θ ≠ NIL	Put Z2 := IMPLY(TYPELIST, H, Aθ)	
7.3	Z2 ≡ NIL		NIL
7.4	Z2 ≡ (θ2 TY2)		(θ •θ2 TY2)
7E.	B ≡ (A→a=b)	Put Z := HOA(a=b, C)	
7E.1	Z ≡ NIL	Go to 7LE	
7E.2	Z ≡ (θ TY1)	Put Z2 := IMPLY(TYPELIST, H, Aθ)	
7E.3	Z2 ≡ NIL		NIL
7E.4	Z2 ≡ (θ2 TY2)		(θ •θ2 (TY1 ∨ TY2))

[4] In case Z2 ≡ NIL it repeats Rule 6 (once) with D ∧ A instead of A ∧ D. If on this second time Z2 ≡ NIL, then (θ TY1) is returned.

(table continued)

7LE.	B ≡ (A→a<b)	Put A' := SET-TYPE(a<b) Let TY' be the updated TYPELIST	
7LE.1	TY' ≡ TYPELIST	Go to 8	
7LE.2	TY' ≠ TYPELIST	Put Z := IMPLY(TY', H, C)	
7LE.3	Z ≡ NIL		NIL
7LE.4	Z ≡ (θ TY1)	Put Z2 := IMPLY(TYPELIST, H, Aθ)	
7LE.5	Z2 ≡ NIL		NIL
7LE.6	Z2 ≡ (θ2 TY2)		(θ ·θ2 (TY1 v TY2))

TABLE II-T. HOA rule changes - TYPELIST version. HOA has arguments (B, C, HL), where B is the hypothesis and C is the conclusion. We ignore HL here.

The other rules of IMPLY and HOA should be changed similarly, always changing an output

$$\theta$$

to

$$(\theta \text{ NIL}).$$

These changes are best explained by the use of examples.

In the following proofs, the theorem label (X h1) is used to indicate that the first hypothesis is being used to try to prove the subgoal (X). Similarly for (X h2), etc. Also the label (X h2 H) is used to indicate that, after backchaining on the second hypothesis (see Rule H7), it is now trying to prove the hypothesis of the second hypothesis, etc.

Ex. 8.

$$(k\leqslant 3 \wedge (k\leqslant 1 \rightarrow C) \wedge (2\leqslant k\leqslant 3 \rightarrow C) \rightarrow C)$$

(1)	$(\{k: 0\ 3\} \wedge \overbrace{(k\leqslant 1 \rightarrow C)}^{\alpha} \wedge \overbrace{(2\leqslant k \wedge k\leqslant 3 \rightarrow C)}^{\beta} \Rightarrow C)$	I7
(1 h1)	$(\{k: 0\ 3\} \wedge (k\leqslant 1 \rightarrow C) \Rightarrow C)$	H6
(1 h1 H)	$(\{k: 0\ 3\} \wedge \alpha \wedge \beta \Rightarrow k\leqslant 1)$	H7, 7.2
	SET-TYPE(~(k≤1)), 2≤k	I11
	TY' = {k: 2 3}, has no contradiction	
	Returns (T {k: 2 3}) for (1 h1 H)	I11.2
	and for (1 h1)	H7.4
(1 h2)	$(\{k: 2\ 3\} \wedge \beta \Rightarrow C)$	H6.4
(1 h2 H)	$(\{k: 2\ 3\} \wedge \alpha \wedge \beta \Rightarrow 2\leqslant k \wedge k\leqslant 3)$	H7, 7.2

(1 h2 H1) $(\{k: 2\ 3\} \wedge \alpha \wedge \beta \Rightarrow 2 \leqslant k)$ I4

SET-TYPE($\sim(2 \leqslant k)$), $k \leqslant 1$ I11
TY' = $\{k: 2\ 1\}$, has a contradiction
Returns(T NIL) I11.1

(1 h2 H2) $(\{k: 2\ 3\} \wedge \alpha \wedge \beta \Rightarrow k \leqslant 3)$ I4.2

SET-TYPE($\sim(k \leqslant 3)$), $4 \leqslant k$
TY' = $\{k: 4\ 3\}$, has a contradiction
Returns(T NIL) I11.1
Returns(T NIL) for (1 h2 H) I4.2
Returns(T NIL) for (1 h2) I7.4
Returns(T NIL) for (1) H6.6

Thus the theorem is true.

Ex. 9

$(2 \leqslant n)$
$\wedge \forall m(2 \leqslant n \wedge 1 \leqslant m \wedge m \leqslant 1 \rightarrow A[m] \leqslant A[2])$
$\wedge \forall k(k \leqslant n \wedge 2 \leqslant k \rightarrow A[k] \leqslant A[k+1])$
$\rightarrow \forall K(K \leqslant n \wedge 1 \leqslant K \rightarrow A[K] \leqslant A[K+1])$

(1) $(2 \leqslant n \wedge \overbrace{(2 \leqslant n \wedge 1 \leqslant m \wedge m \leqslant 1 \rightarrow A[m] \leqslant A[2])}^{\alpha}$
$\wedge \overbrace{(k \leqslant n \wedge 2 \leqslant k \rightarrow A[k] \leqslant A[k+1])}^{\beta}$
$\rightarrow (K \leqslant n \wedge 1 \leqslant K \rightarrow \overbrace{A[K] \leqslant A[K+1]}^{\gamma}))$

n and K are skolem constants

(1) $(\overbrace{[\{K: 1\ n\}\ \{n: \max(2,K)\ \infty\}]}^{TY} \wedge \alpha \wedge \beta \Rightarrow A[K] \leqslant A[K+1])$ I7

(1 h1) $(\alpha \Rightarrow \gamma)$ Returns NIL H6

(1 h2) $(\beta \Rightarrow \gamma)$ H6.1
$(A[k] \leqslant A[k+1] \rightarrow A[K] \leqslant A[K+1]),\ \{K/k\}$ H7

(1 h2 H) $(TY \wedge \alpha \wedge \beta \Rightarrow K \leqslant n \quad 2 \leqslant K)$ H7.2

(1 h2 H1) $(TY \wedge \alpha \wedge \beta \Rightarrow K \leqslant n)$ I4
SET-TYPE($\sim(K \leqslant n)$), $n \leqslant K-1$ I11
TY' = $[\{K: n+1\ n\}\ \{n: K\ K-1\}]$,
has a contradiction, so returns (T NIL) I11.1

(1 h2 H2) $(TY \wedge \alpha \wedge \beta \Rightarrow 2 \leqslant K)$ I4.2
SET-TYPE($\sim(2 \leqslant K)$), $K \leqslant 1$ I11
TY'' = $[\{K: 1\ \min(1,n)\}\ \{n: \max(2,K)\ \infty\}]$
TY'' = $[\{K: 1\ 1\}\ \{n: 2\ \infty\}]$

Here min(1,n) is converted automatically to 1, because it deduces that

$$n \geqslant K \geqslant 1.$$

TY″ has no contradiction but the program detects $\{K: 1\ 1\}$ in TY″ and therefore replaces K by 1 in H, C, and TY″, (and in γ for (1 h1) below). Thus (A[K]⩽A[K+1]) becomes (A[1]⩽A[2]) and TY″ becomes

$$TY''' = [\{K: 1\ 1\}\ \{n: 2\ \infty\}].$$

and the program continues

	Returns (T TY‴) for (1 h2 H2)	I11.3
	Returns (T TY‴) for (1 h2 H)	H7.4
	Returns (K/k TY‴) for (1 h2)	I4.4
(1 h1)	(TY‴ ∧ α ⇒ A[1]⩽A[2])	H6.4 and footnote 4
	(A[m]⩽A[2] ⇒ A[1]⩽A[2]), 1/m	H7
(1 h1 H)	(TY‴ ∧ α ∧ β ⇒ 2⩽n ∧ 1⩽1 ∧ 1⩽1)	H7.2
(1 h1 H1)	(TY‴ ∧ α ∧ β ⇒ 2⩽n)	I4
	SET-TYPE(~(2⩽n)), n⩽1	I11
	TY⁗ = [{K: 1 1} {n: 2 1}]	
	Returns (T NIL) for (1 h1 H1)	I11.1
(1 h1 H2)	(TY‴ ∧ α ∧ β ⇒ 1⩽1 ∧ 1⩽1)	
	Returns(T NIL) by REDUCE	
	Returns(T NIL) for (1 h1 H)	I4.3
	Returns(1/m NIL) for (1 h1)	H7.4
	Returns((K/k 1/m) NIL)	H6.6

Thus the theorem is true.

It can be seen from these examples that the new TYPELIST TY′ which is returned as

$$(\theta\ TY')$$

represents the *cases that have not been proved* by this call to IMPLY or HOA. Thus it represents cases which are still to be proved by further calls to IMPLY. As long as TY′ is not NIL in the returned (θ TY′), then the theorem has not been completely proved. Hence the final return from IMPLY (for the original theorem itself) must be of the form

$$(\theta\ NIL).$$

Else the theorem is considered not to be proved.

Ex. 10.

$$\forall k(k\leqslant 2 \rightarrow A[k]\leqslant A[k+1]) \wedge \forall m(3\leqslant m\leqslant 7 \rightarrow A[m]\leqslant A[m+1]) \wedge \forall n(6\leqslant n\leqslant j \rightarrow A[n]\leqslant A[n+1]) \rightarrow \forall K(K\leqslant j \rightarrow A[K]\leqslant A[K+1])$$

(1) $\overbrace{(k\leqslant 2 \rightarrow A[k]\leqslant A[k+1]}^{\alpha}$
$\wedge \overbrace{(3\leqslant m\leqslant 7 \rightarrow A[m]\leqslant A[m+1])}^{\beta}$
$\wedge \overbrace{(6\leqslant n\leqslant j \rightarrow A[n]\leqslant A[n+1])}^{\gamma}$
$\rightarrow K\leqslant j \rightarrow A[K]\leqslant A[K+1]$

(1)	$(\{K: 0\, j\}\ \{j: K\, \infty\} \wedge \alpha \wedge \beta \wedge \gamma \Rightarrow A[K]\leqslant A[K+1])$	I7
(1 h1)	$(\alpha \rightarrow A[K]\leqslant A[K+1])$ K/k	H6
(1 h1 H)	$(\{K: 0\, j\}\ \{j: K\, \infty\} \wedge \alpha \wedge \beta \wedge \gamma \Rightarrow K\leqslant 2)$	H7, 7.2
	SET-TYPE(~(K≤2)), 3≤K	I11
	TY′ = [{K: 3 j} {j: K ∞}], has no contradiction	
	Returns (T TY′)	I11.3
	Returns (K/k TY′) for (1 h1)	
(1 h2)	$(TY' \wedge (\beta \wedge \gamma \Rightarrow A[K]\leqslant A[K+1])$	H6.4
(1 h2 h1)	$(\beta \Rightarrow A[K]\leqslant A[K+1])$ K/m	H6
(1 h2 h1 H)	$(TY' \wedge \beta \wedge \gamma \Rightarrow 3\leqslant K \wedge K\leqslant 7)$	H7, 7.2
(1 h2 h1 H1)	$(TY' \wedge (\beta \wedge \gamma) \Rightarrow 3\leqslant K)$	I4
	SET-TYPE(~(3≤K)), K≤2	I11
	TY′ = [{K: 3 min(2,j)}], has a contradiction	
	Returns (T NIL)	I11.1
(1 h2 h1 H2)	$(TY' \wedge (\beta \wedge \gamma) \Rightarrow K\leqslant 7)$	I4.2
	SET-TYPE(~(K≤7)), 8≤K	I11
	TY″ = [{K: 8 j} {j: K ∞}] has no contradiction	
	Returns (T TY″)	I11.3
	Returns (T TY″) for (1 h2 h1 H)	I4.4
	Returns (K/m TY″) for (1 h2 h1)	H7.4
(1 h2 h2)	$(TY'' \wedge \gamma \Rightarrow A[K]\leqslant A[K+1])$ K/n	H6.4
(1 h2 h2 H)	$(TY'' \wedge \gamma \Rightarrow 6\leqslant K \wedge K\leqslant j)$	H7, 7.2

(1 h2 h2 H1) (TY'' $\wedge$ γ $\Rightarrow$ $6 \leqslant K$) I4
SET-TYPE($\sim(6 \leqslant K)$), $K \leqslant 5$ I11
TY''= [{K:8 min(5,j)} {j: K ∞}], has a contradiction
Returns (T NIL) I11.1

(1 h2 h2 H2) (TY'' $\wedge$ $\gamma \Rightarrow K \leqslant j$) I4.2
SET-TYPE($\sim(K \leqslant j)$), $j+1 \leqslant K$ I11
TY'' = [{K: max(8,j+1) j} {j: K K-1}], has a contradiction
Returns (T NIL) I11.1
Returns (T NIL) for (1 h2 h2 H) I4.4
Returns (K/n NIL) for (1 h2 h2) H7.4
Returns ({K/m,K/n} NIL) for (1 h2) H6.6
Returns ({K/k,K/m,K/n} NIL) for (1) H6.6

The theorem is proved.

Simplification

The prover utilizes a simplification routine to manipulate algebraic expressions. Its chief function is to put such expressions in canonical form (see (Bledsoe, 1975, p. 27). Many such simplifiers have been programmed (Moses, 1966; Hearn, 1971; Rulifson, *et al.*, 1972; Tyson, 1975; etc.).

Such a routine is crucial in our program for handling TYPELIST and proving assertions about inequalities, because it eliminates the need for adding the field axioms for the real numbers.

Algebraic unification

If k is a skolem variable and b a constant, an ordinary unification algorithm will fail to unify the two expressions: k+2, b+5.

We have augmented our algorithm to handle such arithmetic expressions. In this case the expressions are subtracted and simplified, and then solved for a variable, getting successively:

$$k+2-(b+5) = 0$$
$$k-b-3 = 0$$
$$k=(b+3).$$

Thus (b+3)/k is returned for UNIFY(k+2,b+5).

Similarly, the two expressions

$$B[k+1] = Amax(B,j,k+1)$$
$$A_0[i_0] = Amax(A_0,1,i_0)$$

where B, j, k are variables and A_0, i_0 are constants, are unified as follows: (shown in prefix form)

(UNIFY(= (Array B(+ k 1)) (Amax B j (+ k 1)))
(= (Array A_0 i_0) (Amax A_0 1 i_0)))

(UNIFY (Array B (+ k 1))
(Array A_0 i_0))

(UNIFY B A_0), A_0/B
(UNIFY (+ k 1) i_0) It deduces that
(+ k (+ (-i_0) 1)) = 0, and returns the substitution
(+ i_0 -1)/k

UNIFY Amax(A_0, j, i_0)
Amax(A_0, 1, i_0) 1/j

Returns $\{A_0/B, (i_0-1)/k, 1/j)\}$.

The routine also handles such examples as

UNIFY(A[i_0]+A[j], A[i]+A[j_0]), Easy
UNIFY(A[i_0]+A[j], A[j_0]+A[i_0])

In this last example, even though a canonical form is used there is no assurance that

i_0 precedes j_0

in the canonical ordering, even though i_0 precedes j. Hence the last example and those like it can present problems.

A PROGRAM VERIFICATION SYSTEM

The interactive prover described in (Bledsoe and Tyson, 1975) has been augmented by the features described in the preceding sections, and used as part of a program verification system (Good, *et al.*, 1975). This system is running on the PDP-10 in London's group at the Information Sciences Institute, Marina Del Rey, California, and on the CDC 6600 and the PDP-10 in Good's group at the University of Texas at Austin.

The version at ISI has been augmented extensively by Larry Fagan and Peter Bruell, especially with features to facilitate man-machine interaction.

Both versions are coded in approximately 200 functions in LISP. Two additional subsystems, INFPRINT and XEVAL, are used to augment the prover. INFPRINT is a routine which was coded by Don Lynn at ISI, and which takes an expression in LISP prefix notation and prints it out in (more readable) infix form, with appropriate indentation. XEVAL, which was developed at ISI by Don Good, is a simplification package for handling arithmetic expressions, and also includes the rewrite rules of REDUCE described in (Bledsoe and Tyson, 1975, Table IV). Since the combined code of these programs exceeds the allowed core space for the time-sharing system at UT, a version of UT-LISP has been developed by Mabry Tyson at UT which utilized virtual memory for LISP functions.

Appendix III of (Bledsoe and Tyson, 1975) gives an example of output from

the ISI program.

TYPELIST IN RESOLUTION

The typing and proof by cases procedures described above can also be incorporated into resolution provers if an additional rule is added to resolution, and if the algorithms for simplification, set-type, sup, and inf are included. Also a new algorithm INTERSECT is needed which combines two typelists (see examples below).

Before the start of resolution, after the theorem has been put into clausal form, each literal of the form

$$(a \leqslant b)$$

is converted to a TYPELIST by the algorithm SET-TYPE. Literals of the form

$$\sim(a \leqslant b)$$

are first transformed to $(b+1 \leqslant a)$ before being converted. Thus the new clauses will consist of ordinary literals L and typelist literals T. For example the theorem

$$(x \leqslant 5 \wedge (x \leqslant 1 \rightarrow C) \wedge (2 \leqslant x \wedge x \leqslant 7 \rightarrow C) \rightarrow C)$$

is first converted to ordinary clausal form

1. $(x_0 \leqslant 5)$
2. $(\sim(x_0 \leqslant 1) \vee C)$
3. $(\sim(2 \leqslant x_0) \vee \sim(x_0 \leqslant 7) \vee C)$
4. $\sim C$

and then converted by SET-TYPE to

1. $\{x_0: 0\ 5\}$
2. $\{x_0: 2\ \infty\} \vee C$
3. $\{x_0: 0\ 1\} \vee \{x_0\ 8\ \infty\} \vee C$
4. $\sim C$

Ordinary resolution is performed on non-typelist literals. Any two typelist literals T_1 and T_2 are resolved, by calling

$$\text{INTERSECT}(T_1, T_2).$$

The result is another typelist which is included as a literal of the resolvent. If this resolvent contains a contradiction it is eliminated. For example clauses 1 and 2 above can be resolved on their first literals. Since

$$\text{INTERSECT}(\{x_0: 0\ 5\}, \{x_0: 2\ \infty\} = \{x_0: 2\ 5\},$$

the resolvent of 1 and 2 is

5. $\{x_0: 2\ 5\} \vee C.$

Similarly we get

6.	$\{x_0: 2\ 5\}$	5,4
7.	$\{x_0: 0\ 1\} \vee \{x_0: 8\ \infty\}$	3,4
8.	~~$\{x_0: 2\ 1\}$~~ $\vee \{x_0: 8\ \infty\}$	6,7
9.	~~$\{x_0: 8\ 5\}$~~ or $\square$	8,6.

Since $\{x_0: 2\ 1\}$ and $\{x_0: 8\ 5\}$ contained contradictions they were eliminated. The algorithms SUP and INF are used for this purpose, exactly as described above. Here, for $\{x_0: 2\ 1\}$,

$$\text{SUP}(x_0,\text{NIL}) = 1$$
$$\text{INF}(x_0,\text{NIL}) = 2.$$

Since [2,1] contains no integer we have a contradiction.

The algorithm INTERSECT when applied to type lists

$$(\{x_1: a_1\ b_1\}\ \{x_2: a_2\ b_2\}\ \ldots \{x_n: a_n\ b_n\}),$$
$$(\{x_1: c_1\ d_1\}\ \{x_2: c_2\ d_2\}\ \ldots \{x_n: c_n\ d_n\}),$$

simply intersects the corresponding entries, getting

$$(\{x_1: e_1\ f_1\}\ \{x_2: e_2\ f_2\}\ \ldots \{x_n: e_n\ f_n\}),$$

where $e_i=\max(a_i,c_i)$ and $f_i=\min(b_i,d_i)$.

Consider now Example 10, above.

$$(\forall k(k\leqslant 2 \rightarrow A[k]\leqslant A[k+1])$$
$$\forall m(3\leqslant m \wedge m\leqslant 7 \rightarrow A[m]\leqslant A[m+1])$$
$$\forall n(6\leqslant n \wedge n\leqslant j \rightarrow A[n]\leqslant A[n+1])$$
$$\rightarrow \forall K(K\leqslant j \rightarrow A[K]\leqslant A[K+1])).$$

The ordinary clausal form is

1. $\sim(k\leqslant 2) \vee A[k]\leqslant A[k+1]$
2. $\sim(3\leqslant m) \vee \sim(m\leqslant 7) \vee A[m]\leqslant A[m+1]$
3. $\sim(6\leqslant n) \vee \sim(n\leqslant j_0) \vee A[n]\leqslant A[n+1]$
4. $K_0\leqslant j_0$
5. $\sim(A[K_0]\leqslant A[K_0+1]),$

where K_0 and j_0 are skolem constants and k, m, and n are variables.

The clauses are converted to

1. $\{k: 3\ \infty\} \vee A[k]\leqslant A[k+1]$
2. $\{m: 0\ 2\} \vee \{m: 8\ \infty\} \vee A[m]\leqslant A[m+1]$
3. $\{n: 0\ 5\} \vee [\{n: j_0+1\ \infty\}\{j_0: 0\ n\text{-}1\}] \vee A[n]\leqslant A[n+1]$
4. $(\{K_0: 0\ j_0\}\ \{j_0: K_0\ \infty\})$
5. $\sim(A[K_0]\leqslant A[K_0+1])$

Some of the resolvents of 1-5 are

6.	$\{K_0: 3\ \infty\}$	1,5
7.	$\{K_0: 0\ 2\} \vee \{K_0: 8\ \infty\}$	2,5
8.	$\{K_0: 0\ 5\} \vee [\{K_0: j_0{+}1\ \infty\}\ \{j_0: 0\ K_0{-}1\}]$	3,5
9.	~~$\{K_0: 3\ 2\}$~~ $\vee \{K_0: 8\ \infty\}$	6,7
10.	~~$\{K_0: 8\ 5\}$~~ $\vee [\{K_0: j_0{+}1\ \infty\}\ \{j_0: 0\ K_0{-}1\}]$	8,9
11.	~~$(\{K_0: j_0{+}1\ j_0\}\ \{j_0: K_0$~~ $K_0{-}1\})$ or $\square$	10,4

In each of 9, 10, and 11, a typelist was removed which had a contradiction.

In the above example we did *not* convert the formula $A[k] \leqslant A[k+1]$ to typelist form

$$\{A[k]: 0\ A[k+1]\}.$$

This is controlled in the program by having a list (j_0 K_0 k m n) of those variables and skolem constants which we allow to be typed.

One could allow *all* inequalities to be converted, but in that case a mechanism would need to be provided for unifying expressions when two typelist literals are resolved.

ACKNOWLEDGEMENTS

The work reported here was supported by NSF Grant # DCR74-12886.

REFERENCES

Bledsoe, W.W., Boyer, R.S., and Henneman, W.H. (1972) Computer proofs of limit theorems. *Artificial Intelligence* 3, 1, 27-60.

Bledsoe, W.W. (1974) Program correctness. *Univ. of Texas at Austin, Math. Dept. Memo* ATP-14.

Bledsoe, W.W. (1975) A new method for proving certain Presburger formulae. *Proc IJCAI4.*

Bledsoe, W.W. and Tyson, M. (1975) The U.T. interactive prover. *Univ. of Texas at Austin, Math. Dept. Memo* ATP-17.

Bledsoe, W.W. and Tyson, M. (1975A) Typing and proof by cases in program verification. Univ. of Texas at Austin, Math. Dept. *Memo ATP-15.*

Bundy, A. (1973) Doing arithmetic with diagrams. *Proc. IJCAI3*, pp. 130-138.

Burstall, R.M. (1969) Proving properties of programs by structural induction. *Comp. J.* 12, 1, 47-48.

Cooper, D.C. (1971) Programs for mechanical program verification. *Machine Intelligence* 6 (eds Meltzer, B. and Michie, D.) Edinburgh University Press, Edinburgh.

Good, D.I., London, R.L., and Bledsoe, W.W. (1975) An interactive verification system. *IEEE Trans. on Software Engineering* 1, 59-67.

Hearn, A.C. (1971) Reduce 2: A system and language for algebraic manipulation. *Proc ACM Symp. Symb and Alg. Manipulation 2*, pp. 128-133.

Manna, Z., Ness, S., and Vuillemin, J. (1973) Inductive method for proving properties of programs. *Comm. ACM.*

Moses, J. (1966) Symbolic integration. *M.I.T.–A.I. Memo 97.*

Rulifson, J.F., Derksen, J.A., and Waldinger, R.J. (1972) QA4: A procedural calculus for interactive reasoning. *S.R.I. Tech. Note 13.*

Tyson, M. (1975) An algebraic simplifier. *Univ. of Texas at Austin, Math. Dept. Memo ATP-26.*

Waldinger, R.J. and Levitt, K.N. (1974) Reasoning about programs. *Artificial Intelligence* 5, 3, 235-316.

APPENDIX A

Tables I and II listed below are taken from Section 2 of (Bledsoe and Tyson, 1975). They define IMPLY and HOA, the principal algorithms of the interactive prover described there. The reader is referred to Section 2 of that paper for a full description of them and their use, and several examples.

RULE	IF	ACTION	RETURN
1.	C ≡ "T" or H ≡ "FALSE"		"T"
2.	TYPELIST†		
3.	H ≡ (A ∨ B)[3]		IMPLY(NIL, (A→C) ∧ (B→C))[1]
4.	C ≡ (A ∧ B) (AND-SPLIT)	Put θ := IMPLY(H, A)	
4.1	$\theta \equiv$ NIL		NIL
4.2	$\theta \not\equiv$ NIL	Put λ := IMPLY(H, Bθ)[4]	
4.3	$\lambda \equiv$ NIL		NIL
4.4	$\lambda \not\equiv$ NIL		$\theta \cdot \lambda$[5]
5.	(REDUCE)	Put H := REDUCE(H) Put C := REDUCE(C)	
5.1	C ≡ "T" or H ≡ "FALSE"	Go to 1	
5.2	H ≡ (A ∨ B)	Go to 3	
5.3	C ≡ (A ∧ B)	Go to 4	
5.4	ELSE	Go to 6	
6.	C ≡ (A ∨ B)		HOA(H, C)
7.	C ≡ (A→B)		IMPLY(H ∧ A, B)[6]
7.1	(PROMOTE)		
7.1	Forward chaining		
7.2	PEEK forward chaining		
8.	C ≡ (A↔B)		IMPLY(H, (A→B) ∧ (B→A))
9.	C ≡ (A=B)	Put θ := UNIFY(A, B)	
9.1	$\theta \not\equiv$ NIL		θ
9.2	$\theta \equiv$ NIL	Go to 10	

† See tables of changes to IMPLY in text of paper.

3 By the expression "H ≡ (A ∨ B)" we mean that H has the form "A ∨ B". Rules 4 and 3 are called "AND-SPLITS". See References [2] and [17] of (Bledsoe and Tyson, 1975).

4 If θ has two entries, a/x, b/x with a≠b, then two λs, λ_1 and λ_2, are computed, one for each case, and $\lambda_1 \cdot \lambda_2$[5] is returned for λ.

5 This is just (APPEND θ λ). If θ has an entry a/x and λ has an entry b/x where a≠b, then leave both values in $\theta \cdot \lambda$. For example, if θ=(a/x b/y), λ=(c/x d/z) then $\theta \cdot \lambda$= (a/x b/y c/x d/z).

6 Actually we call IMPLY(OR-OUT(H ∧ A), AND-OUT(B)). See (Bledsoe and Tyson, 1975, page 13).

(table continued)·

10.	C ≡ (~A)		IMPLY(H ∧ A, NIL)
11.	INEQUALITY†		
12.	(call HOA)	Put θ := HOA(H, C)	
12.1	$\theta \not\equiv$ NIL		θ
12.2	$\theta \equiv$ NIL (PEEK)	Put PEEK[7] light "ON" Put θ := HOA(H, C)	
12.3	$\theta \not\equiv$ NIL		θ
12.4	$\theta \equiv$ NIL	Go to 13	
13.	(define C)	Put C' := DEFINE(C)	
13.1	C' ≡ NIL	Go to 14	
13.2	C' ≠ NIL		IMPLY(H, C')
14.	†		
15.	ELSE		NIL

TABLE I. Algorithm for IMPLY(H, C)

7 See (Bledsoe and Tyson, 1975, page 26). The PEEK light is turned off at the entry to IMPLY.

RULE	IF	ACTION	RETURN
1.	Time limit exceeded		NIL
2.	(MATCH)	Put θ := UNIFY(B, C)	
2.1	$\theta \not\equiv$ NIL		θ
2.2	PEEK†		HOA(B, C)
3.	PAIRS†		
4.	C ≡ (A v D) (OR-SPLIT)	Put C' := AND-OUT(C)	
4.1	C' ≠ C		IMPLY(H, C')
4.2	C' ≡ C	Put θ := HOA(B ∧ ~D, A)[8]	
4.3	$\theta \not\equiv$ NIL		θ
4.4	$\theta \equiv$ NIL		HOA(B ∧ ~A, D)[8]
5.1	C ≡ (A→D)		IMPLY(B, C)
5.2	C ≡ (A ∧ D)		IMPLY(B, C)
6.	B ≡ (A ∧ D)	Put θ := HOA(A, C)	
6.1	$\theta \not\equiv$ NIL		θ
6.2	$\theta \equiv$ NIL		HOA(D, C)
7.	B ≡ (A→D)	Put θ := ANDS(D, C)‡	

† See (Bledsoe and Tyson, 1975, Section 4).
8 In Step 4.2, the "~" in (~D) is pushed to the inside; e.g. ~(~P) goes to P, and ~(P→Q) goes to P ∧ ~Q. If D contains no "~" or "→" then (~D) is omitted and the call is made HOA(B,A). Similarly in Step 4.4.
‡ ANDS is explained in (Bledsoe and Tyson, 1975, page 11).

	(Back-chaining)		
7.1	$\theta \equiv$ NIL	Go to 7E	
7.2	$\theta \not\equiv$ NIL	Put λ := IMPLY(H, Aθ)[4]	
7.3	$\lambda \equiv$ NIL	Go to 8	
7.4	$\lambda \not\equiv$ NIL		$\theta \cdot \lambda$
7E.	B ≡ (A → a=b)	Put θ := HOA(a=b, C)	
7E.1	$\theta \equiv$ NIL		NIL
7E.2	$\theta \not\equiv$ NIL	Put λ := IMPLY(H, Aθ)[4]	
7E.3	$\lambda \equiv$ NIL	Go to 8	
7E.4	$\lambda \not\equiv$ NIL		$\theta \cdot \lambda$
8.	B ≡ (A↔D)		HOA((A→D) ∧ (D→A), C)
9.	B ≡ (a=b)	Put Z := MINUS-ON(a,b)	
9.1	Z ≡ 0		NIL
9.2	Z is a number		T
9.3	Z is not a number	Put a' := CHOOSE(a,b) Put b' := OTHER(a,b) § Put H' := H(a'/b'), C' := C(a'/b')	IMPLY(H', C')
10.	B ≡ (A v D)		IMPLY(B, C)
11.	B ≡ ~A		IMPLY(H, A v C)[9]
12.	ELSE		NIL

TABLE II. Algorithm for HOA(B, C)

§ See (Bledsoe and Tyson, 1975, page 16).

9 Actually we use AND-PURGE(H,~A) instead of H, which removes ~A from H.

UNIV. EDINBURGEN EX BIBL

PROBLEM-SOLVING AND DEDUCTION

3

Practical and Theoretical Considerations in Heuristic Search Algorithms*

Ira Pohl

Information Sciences
University of California at Santa Cruz

CONTENTS

1. The need for heuristics

   Discussion of the TSP (traveling salesman problem) as NP, implying that a satisfactorily efficient algorithm will rely heavily on heuristics. A simple NP reduction between the Hamilton path problem and the TSP problem will be proved.

2. A brief history of heuristic search programs

   Especial emphasis on the Michie-Nilsson-Pohl model of heuristic search. In the discussion of efficiency, a new condition of the *monotone property* will replace the Nilsson consistency assumption. It will be proved that this condition is sufficient for all purposes that consistency is used for, and in practice is easier to check.

3. Error analysis as a principal theoretical tool

   The importance of this concept. Principles of worst case error analysis will be applied to tree spaces. This discussion will extend to an analysis of bidirectional search. Outstanding research questions.

4. Practical application to a difficult problem

   The Held-Karp bounding function as a heuristic function for TSP. The principal of *dynamic weighting* in heuristic search.

5. Miscellany

   Knowledge representation in heuristic search programs, research questions, other topics...

*This research is supported by NSF grant D CR-74-20460 and was partially supported by ZWO of The Netherlands while the author was at the Vrije Universiteit, Informatica, Amsterdam.

THE NEED FOR HEURISTICS

The Cook (Cook, 1971) results on the hypothesized non-polynomial time complexity of an important class of combinatorial and logical computations–even where these are decidable problems–places a new emphasis on the need to understand heuristic techniques. What has long been a main theme of artificial intelligence–that many difficult problems can only be efficiently attacked by programs using ad-hoc rules systematized by experts knowledgable in the problem domain–has now been given further backing by NP-theory (Aho, Hopcroft, and Ullman, 1974). It is almost universally accepted that heuristics will remain a vital method for the efficient solution of even finite decidable questions.

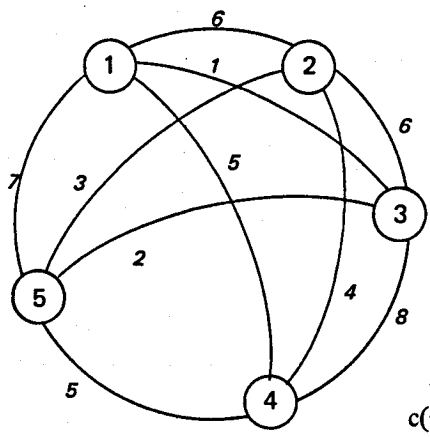
One such practically difficult problem is that of finding optimal solutions to the Traveling Salesman Problem (TSP). As it is a central question in operations research, it has been extensively investigated. The highly developed theory of this problem provides a rich semantics for use in computational solutions. Unlike many toy-problems, such as the Tower of Hanoi puzzle, which are used by artificial intelligence researchers to illustrate the power of their various problem solving systems, the TSP is a problem of known intransigence and importance.

TSP symmetric case

$G = (X,E)$ is a finite undirected graph
$c : E \rightarrow R^+$ where each edge has a positive cost and
$|X| = n$ there are n nodes.
$\Pi(n)$ is the set of all permutations on the n nodes for
$\pi_i(n) \epsilon \Pi(n), \pi_i(n) = (i_1, i_2 \ldots, i_n)$ and

$$c(\pi(n)) = \sum_{j=1}^{n-1} c(i_j, i_{j+1}) + c(i_n, i_1)$$

is the cost of a tour represented by the permutation $\pi_i(n)$. The TSP is to find a permutation $\pi^*(n) \epsilon \Pi(n)$ with minimum cost.



edge	cost
(1,2)	6
(1,3)	1
(1,4)	5
(1,5)	7
(2,3)	6
(2,4)	4
(2,5)	3
(3,4)	8
(3,5)	2
(4,5)	5

$\pi^*(n) = (1,3,5,2,4)$
$c(\pi^*(n)) = 1+2+3+4+5 = 15$

In (Karp, 1972), it is shown that the symmetric Hamilton circuit problem is an NP-problem. The Hamilton circuit problem is to find in a graph a simple circuit which traverses all the nodes of the graph. It is obvious that if there was a program which solved TSP in polynomial time, it could be used to solve the Hamilton circuit problem in polynomial time. What is more surprising is that given a program which on complete graphs finds a solution $\pi'(n)$ to the TSP and $c(\pi'(n)) \leqslant \alpha\, c(\pi^*(n))$ where $\alpha \geqslant 1$ and $\pi^*(n)$ the optimal tour then this program could be used to produce a polynomial time solution to the Hamilton circuit problem.

Proof:

Given $G = \{X,E\}$ generate G' as follows:
if $(x_i,x_j) \in E$ then it is in G' with $c(x_i,x_j) = 1$
if $(x_i,x_j) \notin E$ then it is in G' with $c(x_i,x_j) = \alpha n$.

This recoding of G takes order(n^2) steps.

Let P be a program acting on complete graphs and solving TSP to within α times the optimal tour, then P(G') will generate some solution tour in polynomial time.

If there was a Hamilton circuit in G, then there is a tour in G' of cost n. Tours in G' which included edges not found in G cost at least $n-1+\alpha n$. Therefore when a Hamilton circuit exists in G it will be found by P(G') since only those tours are within α of optimal cost. When no Hamilton circuit exists in G then all tours in G' will be at least cost $\alpha n+n-1$. Thus P(G') decides the question of Hamilton circuits in G.

We can conclude from this that even finding algorithms which efficiently generate reasonable feasible solutions to the TSP in complete graphs will require heuristic programming methods. The list of problems which are known to be NP is already quite large and it is reasonable to conjecture that games such as chess or go, under a suitably devised generalization*, are also in this category.

HEURISTIC SEARCH PROGRAMS

Programs whose main emphasis is to search large spaces by the use of heuristics have been central to artificial intelligence research since its inception. The late 1940's and early 1950's devoted attention to the selective search of spaces defined by such games as chess and checkers (Shannon, 1950; Turing, 1953). The mid-1950's through the early 1960's saw the development of search methods which were *general* in that they could be applied to deductive problems from a variety of domains (Newell, Shaw and Simon, 1960).

*Chess has a finite though large number of legal board positions. In principal all these positions could be catalogued lost, won or drawn. To avoid this, one needs to invent an infinite number of such games which are chess-like but distinct. One possibility would be to make chess boards 8 x n where $n \geqslant 8$.

The mid-1960's produced two striking contributions, both aimed at providing a uniform formal basis for discussing and implementing deductive search. The resolution principle and its associated search strategies (Robinson, 1965) provided one formal model and computational scheme which gained immediate hegemony over prior theorem proving models. The work on graph models of problem domains and the consequent rephrasing of deductive tasks into this language (Doran and Michie, 1966) provided an adequate uniform language to discuss and compare various heuristic search schemes.

So far the 1970's have seen two further themes elaborated; one has been a partial unification of the graph model and resolution model of search processes and a consequent crossfertilization of useful theory and technique between both (Kowalski, 1972; Michie and Sibert, 1974). The second theme, which is the hallmark of progress in much of computer science, is the incorporation of heuristic search techniques as features of high-level programming languages (Hewitt and Smith, 1975).

These formalizations also make it possible to develop a theory of efficiency. While this has become the chief subject of research in theoretical computer science under the banner of complexity-theory, it is still more discussed than pursued in artificial intelligence. The mathematical ugliness of the domains investigated by artificial intelligence is a formidable barrier to obtaining results comparable to those of, for example, the sorting theorist (Pohl, 1975). Nevertheless important advances have occurred (Minsky and Papert, 1969) in artificial intelligence problem domains and heuristic search efficiency will form the chief topic of discussion in this paper. In heuristic search, these questions are deeply related to the accuracy of the heuristic functions. So that search efficiency is bound up with the analysis of *error* in the heuristics used for guiding search. In one sense the problem of *knowledge representation* by which I mean the *conversion of human expertise into machine usable form*, is reformulated as a problem of accuracy and efficiency. The program must have an appropriate representation of the problem and a search function which is accurate with respect to this representation. Either of these requirements by itself is not enough.

ON REPRESENTING PROBLEM DOMAINS AND SEARCH ALGORITHMS

In order to discuss these issues more concretely, we make use of a graph model of heuristic search and a program which can search problems represented in this model.

A problem space is a locally finite directed graph G.

$G = (X,E)$

$X = \{x_1, x_2, \ldots.\}$, X is the set of nodes and can be infinite

$E = \{(x_i,x_j) \mid x_i,x_j \in X \text{ and } x_j \in \Gamma(x_i)\}$, E is the set of edges and can be infinite, however $|\Gamma(x_i)|$ must be finite.

Γ is the successor mapping and if $x \in \Gamma(y)$ then
$y \in \Gamma^{-1}(x)$ the predecessor mapping

In using directed graphs to represent problem domains a state description is associated with each node. The successor mapping defines the structural character of the problem space, identifying for each node its immediate neighbors. A problem consists of some node (or set of nodes) which is the initial node and another node (or set of nodes) which is the terminal node. A solution to a particular problem is a path in G from the (an) initial node to the (a) terminal node. One thinks of a game such as the 15-puzzle where a state description represents the placement of pieces and the successor relation produces the legal moves.

An important sub-category of problems treated by this model associates positive cost with each edge. A solution is then called optimal if it is the least costly path. Frequently in combinatorial problems an optimal or near optimal solution is required.

$c : E \rightarrow R^+$ costs of the edges
$\mu(s,t) = (s = x_1, x_2, \ldots, x_k = t)$ is a path from s to t then

$$c(\mu(s,t)) = \sum_{i=1}^{k-1} c(x_i, x_{i+1})$$

It is sometimes of interest to treat c as a computational cost in producing a state. When this is the case the default understanding is that c is unity, i.e., the cardinality metric.

The algorithm that is presented here is the Heuristic Path Algorithm (HPA). It is one of a number of such algorithms (Nilsson, 1971; Michie and Ross, 1970; Slagle and Bursky, 1968); and is selected from this list because it is sufficiently general so that key issues of heuristic search can be illustrated in terms of it; and because the author and others have developed the work on error and efficiency in this schema (Pohl, 1970a).

HPA description

$G = (X,E)$, the problem graph
$c : E \rightarrow R^+$ = edge costs - most often 1
s = start or initial node, t = terminal or goal node
$g(x)$ = the cost of the path found by HPA from s to x
$h(x)$ = the estimated cost of a path from x to t
$w(x)$ = a weighting term, $0 \leqslant w(x) \leqslant 1$
$f(x) = (1-w(x))g(x) + w(x)h(x)$, the evaluation function
father(x) = the immediate predecessor x as found by HPA
S = set of nodes already visited by HPA; called the expanded or closed nodes.
$\overline{S}$ = set of nodes that are the immediate successors of nodes in S but are not themselves in S; called the open or candidate nodes.

1. Place s in S and generate $\Gamma(s)$ placing these nodes in $\overline{S}$. For $x \in \Gamma(s)$, father(x) = s, g(x) = c(s,x), and f(x) = (1-w(x)) c (s,x) + w(x) h (x).
2. Select $n \in \overline{S}$ such that f(n) is a minimum. If there are ties select among the nodes that have minimum f(n), any node with maximum g(n).
3. Place n in S and for all $x \in \Gamma(n)$ such that x is not in S compute g(x) = g(n) + c(n,x) and f(n) = (1-w(x)) g (x) + w(x) h(x). If x is not in $\overline{S}$ or if it is and the new computation of f(x) is less than the old, place x in $\overline{S}$ with value f(x) and father(x) = n.
4. If n = t halt, otherwise go back to step 2.

HPA upon successfully finding the terminal node has constructed the deductive chain t, father(t), father(father(t)),. . ., father(. . .father(t). . .) = s, with path cost g(t). HPA's total computational effort in finding this path is proportional to the cost of building the set S. In the normal case where unit edge costs are used, search effort is proportional to |S|. This simple observation is critical to evaluating the various forms of search procedure that are proposed.

HPA is an algorithm whose ancestry combines combinatorial search procedures and artificial intelligence search procedures. When w(x) = 0 and the graphs being searched have a cardinality metric then the algorithm is similar to the Moore maze searching algorithm. When more general costs are allowed, it is a form of the Dijkstra two-point shortest path algorithm. When w(x) = 1, then the algorithm is a form of the original graph traverse of Doran and Michie. The graph traverser inaugurates the linking of combinatorial search algorithms and heuristic search algorithms. Previous artificial intelligence researchers made strenuous efforts to develop terminologies which obscured this important affinity.

The next achievements are a first attempt at a theory of search efficiency that centered around the notion of "admissibility." A search algorithm is admissible if the solution path it finds are of minimum cost. Hart, Nilsson and Raphael (Hart, *et al.*, 1968) devised the algorithm A* which is a form of HPA restricted to w(x) = 0.5. They restricted their attention to heuristic functions which were guaranteed to return a lower bound on the remaining cost of a solution path. A* using such a heuristic is provably guaranteed to find a least costly solution path. Two remarks are in order: (i) h(x) = 0 is trivially a lower bound and gives us back Dijkstra's algorithm; (ii) A* is a form of the branch-and-bound principle which is a key method in enumerative combinatorial programming.

The above developments led to a natural generalization of these methods HPA (Pohl, 1969,1970a). This formulation revealed two critical questions: (i) what are the best values for w(x); (ii) how is search efficiency dependent on the accuracy of h. The first question could not be posed in terms of the predecessors to HPA. The second question received significant practical testing in the work of the graph traverser and a limited theoretical treatment within the context of A* performing admissible searches. The proper formulation of a theory of error in heuristics and its effect on search performance was a chief accomplishment of the HPA work. This issue continues to be, along with the question of incor-

porating high level planning into search procedures, the critical research questions in heuristic search.

ADMISSIBILITY AND SEARCH EFFICIENCY

To discuss search efficiency and its relation to the accuracy of the heuristic information employed in directing search, we introduce some further terminology.

K(m,n) = the cost of an optimal path from m to n
$h^*(n) = K(n,t)$, to be called the *perfect* heuristic
If $\forall_{x \in X}(h(x) \leq h^*(x))$ then h is an *admissible* heuristic
If $\forall_{x,y \in X}(h(x)-h(y) \leq k(x,y))$ and $h(t)=0$ then h is a *consistent* heuristic.

Theorem

If HPA with $w(x) = \frac{1}{2}$ is used with a consistent heuristic, then all nodes included in S will have $g(x) = k(s,x)$. (Hart, Nilsson, and Raphael, 1968).

Corollary

The above scheme will compute an optimal path to a terminal node t, hence it is an admissible search algorithm.

A* is described in a way that allows nodes in set S to be re-evaluated and if necessary placed back in $\bar{S}$. Under this additional computational burden, A* is admissible if its heuristic is admissible. This is not the case for HPA with $w(x) = \frac{1}{2}$ which for admissibility requires that the heuristic used be consistent (exercise: show how HPA with $w(x) = \frac{1}{2}$ and h admissible can lead to a non-optimal solution path).

Consistency is a strong constraint and whereas $h(x) = 0$ trivially satisfied consistency, it may be hard to demonstrate for more informed heuristics. An alternate property which is more easily checked is the *monotone* criterion.

A heuristic function satisfies the monotone criterion if for all $x \in X$ and $y \in \Gamma(x)$, $0 \leq h(x)-h(y) \leq c(x,y)$ and $h(t) = 0$.

Theorem

HPA with $w(x) = \frac{1}{2}$ and h admissible and monotone will only include nodes in S for which $g(x) = k(s,x)$.

Proof

By induction on the number of nodes placed in set S.

$n = 1$: The first node placed in S is s with $g(s) = 0$; so the theorem is trivially true in this case.

Assume true for $k < n$ to prove for n.

Let x be the nth node placed in set S and $g(x) > k(s,k)$ and the

path from s to x is $s = x_1, x_2, \ldots, x_r = x$. There is a shortest path from s to x namely $s = y_1, y_2, \ldots, y_p = x$. Let y_ℓ be the last node along this path which is in S. So $\ell < p$, otherwise x would have been placed in S with its optimal cost. The evaluation function for $y_{\ell+1}$ is

$$f(y_{\ell+1}) = \tfrac{1}{2}\{(g(y_\ell) + h(y_{\ell+1}) + c(y_\ell, y_{\ell+1})\}$$

claim: $f(y_{\ell+1}) < f(x)$

First we show that $f(y_k) \leq f(y_{k+1})$.

By the monotone criterion $h(y_k) - h(y_{k+1}) \leq c(y_k, y_{k+1})$

$$f(y_{k+1}) = \tfrac{1}{2}\{g(y_k) + h(y_{k+1}) + c(y_k, y_{k+1})\}$$
$$f(y_k) = \tfrac{1}{2}\{g(y_k) + h(y_k)\}$$

$2[f(y_k) - f(y_{k+1})] = h(y_k) - (y_{k+1}) - c(y_k, y_{k+1}) \leq 0$ and so

$f(y_k) \leq f(y_{k+1})$ and, by transitivity,
$f(y_{\ell+1}) \leq f(y_p) < f(x_r)$.

Therefore the node $y_{\ell+1}$ will be included in S before x, a contradiction, x must be included in S only when $g(x) = k(s,x)$.

ERROR ANALYSIS

The computational effort required by HPA is proportional to the number of times the inner loop is executed, which is equivalent to $|S|$. This set upon successful termination of HPA must be at least the size of the number of nodes on the shortest solution path. Indeed this performance is possible when HPA uses h^* with $w(x) \geq \frac{1}{2}$ (Pohl, 1970a). The usual situation is less fortunate as for most problems the heuristics will be in error. We would like a precise basis for formulating the concept of error and a methodology for investigating its consequent effect on search. This theory was first elaborated in (Pohl, 1969; Pohl, 1970b).

We analyze search behavior by adopting the worst case norm for measuring the efficiency of our algorithms. Namely, given all possible inputs conforming to a problem statement what is the most effort the algorithm will require for any input in this set. This is the most widely applied norm in the complexity literature. While in many instances this norm is not as relevant as average performance, it allows a significant reduction in the difficulty of proving results.

Worst case analysis often is performed with the aid of an oracle or adversary strategy (Knuth, 1973; Kirkpatrick, 1974). Since the efficiency of an algorithm depends on how certain tests are decided, the adversary attempts to provide results which are consistent with the problem constraints but which maximally degrade performance.

We restrict our attention to heuristic functions of *bounded error* $\epsilon = 0,1,\ldots,$ as applied to tree domains in search of solution paths of length k.

$$\forall_{x \epsilon X}\{h^*(x) - \epsilon \leqslant h(x) \leqslant h^*(x) + \epsilon\},$$

h(x) is said to be of bounded error ϵ.

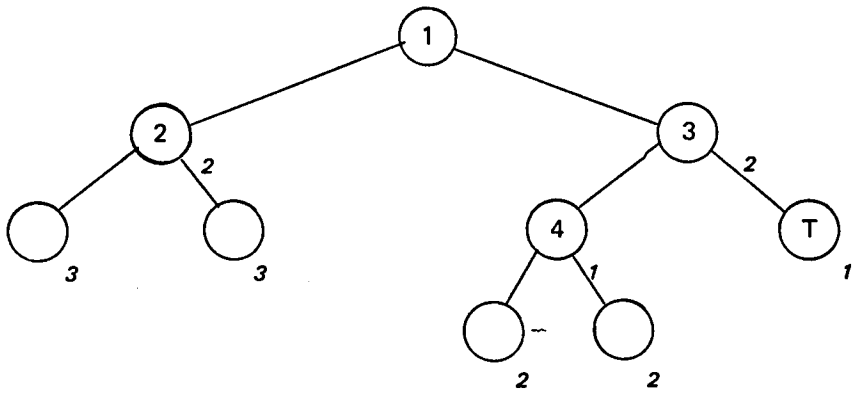
Oracle for HPA used in computing worst-case analysis

If x is along a shortest path, $h(x) = h^*(x)+\epsilon$, otherwise $h(x) = h^*(x) - \epsilon$.

It can be proved that this oracle leads to worst-case performance for HPA acting on tree domains (Pohl, 1970b).

Example:

HPA will be used on a binary tree domain with $k = 2$, $\epsilon = 1$, and $w(x) = 1$.



The goal node is marked by the "T" and the start node is node 1. The internal numbering represents the order in which nodes are placed in S. The external numbering represents the oracles response to evaluating h(x).

If the above analysis is extended to the general case with HPA using $w(x) = 1$, pure heuristic search, then all nodes $2\epsilon - 1$ off the shortest path will be included in S (Pohl, 1970b, Theorem 8). So for binary trees, pure heuristic search in the worst case expands $2^{2\epsilon-1}$ K+1 nodes, $\epsilon \geqslant 1$, where k is the length of the solution path. A similar analysis of HPA with $f(x) = \frac{1}{2}(g(x)+h(x))$ has HPA expanding 2^{ϵ} K+1 nodes in the worst case.

The derivation of these two results reveals an unsuspected theoretical advantage in favor of utilizing the g(x) term in the evaluation function. Until the time of this result and empirical results backing up this relationship, all artificial intelligence researchers who confined their attention to finding non-optimal cost solutions to deductive problems believed that pure heuristic search was the most

efficient mechanism. Else, they had not given any thought to the problem.

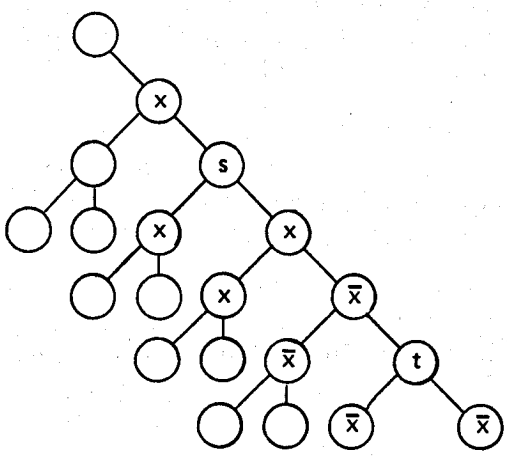
HPA and like search algorithms can be amended to search problem spaces bi-directionally (Pohl, 1971). HPA is a uni-directional algorithm searching in the forward direction, out from s. It is often possible when t is explicitly given and $\Gamma^{-1}(x)$ the predecessor relation is available to search back from t looking for s. In symmetric graphs this is accomplished by renaming s as t and vice-versa. It is possible to combine both a forward and a backward search by i) using a choice rule to decide on which direction to search on a given iteration and ii) stopping when some node n is found in the intersection of the two searches.

The motivation for doing this is clear–search effort grows exponentially with the depth of search; two searches roughly half the depth of one longer search provides exponential savings easily offsetting the cost of the extra administrative work. This is in fact the case for the ordinary shortest path problem (Pohl, 1969) where no heuristic information is used. However, a number of experiments with bi-directional serach using heuristic information have led to *worse* performance than uni-directional search.

It is of interest to see if error analysis of the bi-directional case can point out the difficulties. We embed the solution path in a portion of sufficiently large binary tree and perform worst case analysis. We will do an example with pure heuristic search and with the bi-directional choice rule of 'alternating on each iteration'.

Example:

$k = 3, \epsilon = 1, w(x) = 1$ (both directions)



s - start node forward direction
t - terminal node - start node backward direction
x - nodes closed in the forward direction
$\bar{x}$ - nodes closed in the backward direction

Under the above assumptions, bi-difectional heuristic search expands a few more nodes than uni-directional search–those nodes which would ordinarily be successors of t would not be examined by uni-directional search. Note, worst case analysis with respect to functions of bounded error leads to search effort proportional to path length multiplied by a term exponentially related to error. Thus bi-directional search does not yield exponential savings as growth is not exponential in terms of path length. So heuristic search with reasonably effective search functions in the worst case does not benefit from bi-directional methods.

Let us perform our worst case analysis with respect to a different bound on heuristic error. We will say that a heuristic function h(x) has *relative error* δ if

$$\forall_{x \in X} \{h^*(x)(1-\delta) \leqslant h(x) \leqslant h^*(x)(1+\delta), 0 \leqslant \delta < 1\}.$$

In many ways this is more realistic than bounded error. It conforms to the common sense expectation that estimating bigger distances will lead to larger error.

Applying a worst case analysis within these bounds on the heuristic function gives results which even more strongly than in the bounded error case favor uni-directional methods. In the binary tree space of the previous example define T_j as the subtree of nodes which hang off the jth node along the solution path and which are expanded by HPA in the worst case.

$$\text{Total number of nodes expanded} = \sum_{j=o}^{k} |T_j| + k + 1$$

By the worst case assumptions (again may be derived from an oracle which maximally penalizes nodes along the solution path and maximally favors nodes off the solution path) the height of T_j is

$$\text{height } T_j = \left\lfloor \frac{2K\delta}{1-\delta} - 2 - \frac{1}{1-\delta} \right\rfloor$$

$$|T_j| = 2^{(\text{height } T_j + 1)} - 1.$$

So in the case of relative error, path length enters in exponentially as well as error. Bi-directional search in tree spaces under these assumptions is roughly *twice* as expensive as uni-directional search. This theoretical result is in surprising agreement with observed experiment (Pohl, 1971).

We introduce some further notation which is needed in discussing bi-directional search.

$h_s(x)$ = estimator of k(x,t)
$h_t(x)$ = estimator of k(s,x)
$g_s(x)$ = distance found in forward search from s to x
$g_t(x)$ = distance found in backward search from t to x
h(x,y) = estimator of k(x,y)

In the just concluded argument bi-directional search uses in the forward direction $h_s(x)$ and in the backward direction $h_t(x)$ to guide search. Each search is being guided to the opposite end-point. One form of improvement suggested in (Pohl, 1971) was for both searches to aim at a common midpoint. Or in the absence of a convenient means of formulating a hypothesized midpoint, the evaluation function could be computed with respect to newly selected end-points. These could be the pair (s',t') such that $g_s(s') + g_t(t') + h(s',t')$ is a minimum among all pairs of nodes $s' \epsilon \bar{S}$, $t' \epsilon \bar{T}$. Recent work has been performed according to these suggestions (de Champeaux and Sint, 1974). The results were promising in that bi-directional methods employing these ideas on the 15-puzzle typically found shorter solution paths than the use of the unidirectional method with the same heuristics. However, the number of nodes expanded was comparable and this meant that the uni-directional method was more efficient as it does not incur the additional administrative costs that bi-directional search requires; including the cost of evaluating h(x,y) for all node pairs in the open sets.

The above remarks suggest that there may be an appropriate number of iterations in a single direction before switching to the opposite direction. This would be related to the accuracy of the heuristic function. It would seem that a more accurate heuristic function could take advantage of more frequent alternation of search direction and updating of endpoint pairs. This point is partially supported by the experiments with "look-ahead" search (Rosenberg and Kestner, 1972). Instead of computing $\Gamma(x)$, they computed $\Gamma^k(x)$ and only then computed the evaluation function for open nodes. They found that in a number of instances the additional effort of computing $\Gamma^k(x)$ was offset by the improvement in choice of which next node to expand, i.e., look-ahead acts to improve the accuracy of the heuristics being used. It is worth pointing out that this technique resembles the GT-4 utilization of macro-moves (Michie, 1971; Ross, 1973).

PRACTICAL APPLICATION TO A DIFFICULT PROBLEM

The TSP has already been described and we now return to it as a case study of incorporating expertise into HPA. As it has already been proven that finding solutions to TSP which are within some known bound of optimal is NP, we are justified in using heuristic procedures.

To try and avoid confusion between the problem space graph and the TSP graph the latter will be written in italics, e.g., $G(X,E)$ with edge costs $c(x_i,x_j)$. A node x in the problem graph will contain as state description the two sets of edges A,R with meaning:

> $e_i \epsilon$A means $e_i \epsilon E$ and e_i is to be included in any tour produced by a descendant of x
>
> $e_j \epsilon$R means $e_j \epsilon E$ and e_j is *not* to be included in any tour produced by a descendent of x

$A \cap R = \phi$ and furthermore there are no three edges of E in A such that all have as endpoint the same node of X.

The successor relation $\Gamma(x)$ will either create a single new node with $A(y) = e \cup A(x)$ a tour or the two nodes y and y′ with

$$A(y) = \{e\} \cup A(x), \qquad R(y) = R(x)$$
$$A(y') = A(x), \qquad R(y') = \{e\} \cup R(x)$$

e is an edge of E not in $A(x) \cup R(x)$ and is chosen by a "suitably expert" successor relation. By this we mean that the edge to be the next edge included in the partial tour A is somehow a reasonably intelligent choice. Any node which represents a tour is a terminal node; the first such node included in S terminates HPA.

The heuristic function to be used is the minimum spanning 1-tree (Held and Karp, 1971). A tree is a connected graph without cycles. A minimum spanning tree is the minimum cost tree over all trees which are subgraphs of G and include all nodes of G. A 1-tree is a tree over the node set $\{x_2, x_3, \ldots, x_n\}$ plus two distinct edges with x_1 as their endpoint. A minimum (spanning) 1-tree is readily computed by computing a minimum spanning tree on the node set $\{x_2, x_3, x_4 \ldots x_n\}$ using Dijkstra's or Kruskal's algorithm (Aho, Hopcroft, and Ullman, 1974) and adding the two cheapest edges connected to x_1. Such a computation is order(n^2).

A tour is a minimum 1-tree in which each node has degree two. A minimum 1-tree is a least costly element of a superset of tours, hence it is a lower bound on tours. We compute f(x) where x is associated with A(x) and R(x) in the following manner:

$g(x) = \sum_{e_i \in A(x)} c(e_i)$, the cost of the edges that must be included in a tour.

h(x) = the cost of a minimum 1-tree which includes the edges A(x) and excludes the edges R(x) - g(x)

Alternatively, if y is the father of x and x is a node created by adding e to the set A(y) then $g(x) = g(y) + c(e)$, and if x is a node created by adding e to the set R(y) then g(x) = g(y); i.e., in the former case $c(y,x) = c(e)$ and in the latter case c(y,x) = 0. With these definitions of g and h, it is readily shown that h is monotone and therefore $f(x) = \frac{1}{2}g(x) + \frac{1}{2}h(x)$ is admissible.

The heuristic function just defined is often not accurate, but it is possible by the following gradient technique to improve its accuracy.

Given a real n-vector $\overline{v} = (v_1, v_2, \ldots, v_n)$ we modify the edge costs in G,

$$c_{\bar{v}}(x_i, x_j) = c(x_i, x_j) + v_i + v_j.$$

It is obvious that if $\pi(n)$ is a tour in G of cost $c(\pi(n))$ then it is a tour in G with

respect to edge costs $c_{\bar{v}}$ of cost $c(\pi(n)) + 2\sum_{i=1}^{n} v_i$. So the same tour is optimal with respect to any such modification of costs by a vector $\bar{v}$. Now a tree is not a tour in-so-far as there are nodes whose degree differs from 2. It is desirable to penalize with a large value of v_j a node x_j whose degree in a 1-tree exceeds two. In a like manner it is desirable to reward with a negative value of v_j a node x_j whose degree in a 1-tree is one. This can be done by computing the i + 1st spanning tree from the original costs modified by v^i. This tree has nodes whose degree is d_1^{i+1}, d_2^{i+1},..., d_n^{i+1}. Then compute $\bar{v}^{i+1}$ as the sum $v_k^{i+1} = v_k^i + d_k^{i+1} - 2$. The degrees themselves provide a reasonable means of penalizing nodes. Among the sequence of 1-trees a best estimate is provided by that tree whose cost, in relation to the original edge cost, is maximum. One terminates this iteration when either a 1-tree is a tour or when no further improvement is expected.

To complete our description of HPA using the above functions to solve TSP we must give a rule for deciding which edge will be used for generating successor nodes in the problem graph. It will be that edge, which when removed from the spanning 1-tree computation, causes its recomputation to maximally increase in cost—in effect a form of "killer heuristic." It is also necessary to discuss the values of w(x). Using w(x) = ½ as has been stated gives an admissible algorithm akin to branch-and-bound. Since the h just described is a guaranteed lower bound it can serve no useful purpose to consider values $0 \leqslant w(x) < ½$. However $w(x) > ½$ will lead to possibly non-optimal solutions.

This overweighting of the heuristic term allows us to find a *computational compromise* betwen the two standard *computational catastrophes* of heuristic search procedures. That is if w(x) is ½ (or near ½) we are guaranteed to produce a solution which is optimal (or near optimal) but because the search is too breadth-first we run out of space/time resources. If w(x) is near 1, we have pure heuristic search which in this case is likely to examine few nodes in *G* but is also likely to generate an unacceptably bad tour. We compromise by using a w(x) which balances these two needs.

So far in this discussion w(x) has been a fixed constant in any particular application. The above considerations lead naturally to a dynamic computation of the weighting term. In *dynamic weighting* in relation to admissible heuristic functions we set $w(x) = ½ + \beta$ where x is a direct successor of s. When we are deeper into the problem graph, further away from s, we reduce w(x) to ½. This reduction is proportional to the distance away from S. Dynamic weighting produces better solutions than uniformly overweighting the heuristic function. It is almost as efficient in terms of nodes expanded because it only becomes breadth first as it nears a solution node. Dynamic weighting compensates for the *difficulty-phenomenon* in heuristic search. This is an observed degradation in the accuracy of most heuristics the further away from a terminal node they are

applied. The use of dynamic weighting has led to methods competitive in efficiency with the Lin (Lin, 1965) techniques for TSP, and providing firmer guarantees on the optimality of the resulting solutions (Pohl, 1973). Similar ideas are applicable to game tree searches such as the Harris bandwidth search (Harris, 1973).

Ramblings

The combinatorial explosion (Lighthill, 1973) is readily identified as a chief obstacle to effective problem solving. The two main tools of artificial intelligence which attempt to overcome this problem are heuristic search and the dense utilization of specific knowledge. The chess player who decides on a move typically has selected a "plan" he deems suitable for his position. This plan in the absence of tactical threat may focus his attention on a single piece or more typically an area of the board. Such is the case with knight manoeuvres or with the minority attack. Tactical play being inherently local can be analyzed by a limited exhaustive search. Similarly, the mathematician begins by selecting a method of proof which appears suitable for a particular problem, such as induction or proof by contradiction. His strategy may further involve identifying useful intermediate results–lemmas which he feels tactically capable of solving. In relation to simpler sub-problems he may use locally exhaustive methods of solution.

More generally in artificial intelligence and computer science, we seek to avert the combinatorial explosion in three ways:

(1) detecting efficiently solvable sub-cases
(2) utilizing efficient schemes which give approximate "non-optimal" solutions
(3) utilizing heuristic search techniques to facilitate general searches

Each of these methods is "knowledgable." An instance of (1) is the chess players use of special procedures for the elementary endgames. Often the principle of king opposition selects without search the correct move. An example of (2) is the Lin technique for the TSP problem (Lin, 1965). A tour is improved by determining whether the deletion of 3 edges and their replacement will improve the tour cost. This is continued until a tour which cannot be improved according to this standard is achieved. This is always possible to compute in polynomial time. In chess a technique in this category is to perform trades when ahead in material. Instances of (3) have been the main content of this paper.

It is interesting to note that a main contributor to this theory has recently said (Nilsson, 1974 p. 18), "The problem of efficiently searching a graph has essentially been solved and thus no longer occupies AI researchers." What could be meant by this I am not sure. On the one hand the solution to efficiently solving the two-point path problem has long been known when heuristic information is not involved. On the other hand it is the deepest unsolved problem in

complexity theory to determine whether certain searches in graphs can be done in polynomial time. Furthermore in AI there has been a recent surge in transferring or embodying heuristic search procedures into programming languages (Bobrow and Raphael, 1973). A key difficulty to implementing search procedures implicitly is the inefficiency of any general control structure capable of adminstrating these searches. In that this remains a key research problem for PLANNER-like systems, it is obvious that further progress on heuristic search theory is a necessary precursor to its solution.

As was stated earlier the question of high level planning is one of the central research questions in heuristic search. It has been adequately stressed in any number of forums, and one consequence is the great popularity of planning-languages. It is mildly disconcerting that much of this work ignores or is ignorant of its natural predecessors the various heuristic search systems. The second issue which is central to heuristic search theory is a need to elaborate the results of error analysis. In their description of Merlin (Moore and Newell, 1973) and its comparison to other systems which "understand" knowledge, Moore and Newell say (*ibid.*, p. 50) "The design of Merlin offers nothing fundamental yet to cope with error... we have let the issue of error alone." Now it is clear from their paper that error is not just the use of inaccurate heuristics in that they additionally include other forms of representational error such as the frame problem. Nevertheless while examining a wide number of standard search methods including backtracking, they do not broach the one concrete theory of error for which results exist. The effect of these omissions in the fashionable literature is to underline the significance of these research topics in heuristic search.

I will conclude by stating three principles of heuristic search.

Principle of Utilization

Heuristics are necessary for the solution of difficult problems–even in many finite domains. Hence the process of discovering heuristics is the key task of problem solving.

Principle of Robustness

Heuristics are approximate uses of knowledge–the fact of them being in error affects their use. They should be employed in schemes which can recover from error.

Principle of Adaptability

Heuristics tend to be locally good–the nearer a solution the better. Their most effective use is adaptive (dynamic) with respect to their accuracy.

REFERENCES

Aho, A.V., J.E. Hopcroft and J.D. Ullman (1974) *The Design and Analysis of Computer*

Algorithms, Addison-Wesley, Reading, Mass.

Bobrow, D.G. and B. Raphael (1973) New Programming Languages for AI Research, Tutorial Lecture given at IJCAI 3.

Cook, S.A. (1971) The complexity of theorem proving procedures. *Proc. 3rd Annual ACM Symposium of the Theory of Computing* pp. 151-158.

de Champeaux, D. and L. Sint (1974) An improved bi-directional heuristic search algorithm. *Report* Dec. 1974, Instituut voor Bedrijfs economie en Accountancy, Universiteit van Amsterdam, Amsterdam.

Doran, J. and D. Michie (1966) Experiments with the Graph Traverser program. *Proc. Roy. Soc. A* 294 pp. 235-259.

Harris, L. (1973) The bandwidth heuristic search. *Proc. IJCAI3* pp. 23-29.

Hart, P., N. Nilsson, and B. Raphael (1968), A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Sys. Sci. and Cyber.* 4, 100-07.

Held, M. and R. Karp (1971) The traveling salesman problem and minimum spanning trees–Part II. *Mathematical Prog.* 1, 6-25.

Hewitt, C.E. and B. Smith (1975) Towards a programming apprentice. *IEEE Trans. on Software Engineering*, 1, 26-45.

Karp, R.M. (1972) Reducibility among combinatorial problems, *Complexity of Computer Computations*, (Miller and Thatcher, eds.) Plenum Press, New York, pp. 85-104.

Kirkpatrick, D.G. (1974) *Computer Science Report TR. 74*, University of Toronto, Toronto.

Knuth, D.E. (1973) *Sorting and Searching*, Addison-Wesley, Reading, Mass.

Kowalski, R. (1972) AND-OR graphs, theorem-proving graphs and bi-directional search *Machine Intelligence* 7 (Meltzer and Michie, eds.) Edinburgh University Press, Edinburgh, pp. 167-94.

Lighthill, J. (1973) Artificial Intelligence: A General Survey *Artificial Intelligence: A Paper Symposium*, Science Research Council, London.

Lin, S. (1965) Computer Solution of the traveling salesman problem. *Bell Sys. Tech. J* 44, 10, 2245-2269.

Michie, D. and R. Ross (1970) Experiments with the adaptive Graph Traverser. *Machine Intelligence* 5 (Meltzer, B. and Michie, D., eds.) Edinburgh University Press, Edinburgh, pp. 301-318.

Michie, D. (1971) Formation and execution of plans by machine. *Artificial Intelligence and Heuristic Programming*, (Findler, N.V. and Meltzer, B., eds.) Edinburgh University Press, Edinburgh.

Michie, D. and E.E. Sibert (1974) Some binary derivation systems. *JACM* 21, 175-190.

Minsky, M. and S. Papert (1969) *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, Mass.

Moore, J. and A. Newell (1973) How can Merlin understand? *Report*, Dept. Comp. Sci. Carnegie-Mellon Univ., Pittsburgh.

Newell, A., Shaw, J.C. and H.A. Simon (1960) Report on a general problem-solving program for a computer. *Information Processing: Proc. of the Int. Conf. on Information Processing*, Paris: UNESCO pp. 256-264.

Nilsson, N. (1971) *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York.

Nilsson, N. (1974) Artificial Intelligence. *SRI Artificial Intelligence Center Rep 89*, also given at IFIP Congress 1974, Stockholm.

Pohl, I. (1969) *Bi-directional and Heuristic Search in Path Problems, SLAC Report 104*, Stanford.

Pohl, I. (1970a) Heuristic search viewed as path finding in a graph, *Artificial Intelligence* 1, 193-204.

Pohl, I. (1970b) First results on the effect of error in heuristic search, *Machine Intelligence* 5, (Meltzer, B. and Michie, D., eds.) Edinburgh University Press, Edinburgh, pp. 219-236.

Pohl, I. (1971) Bi-directional search. *Machine Intelligence* 6, (Meltzer, B. and Michie, D., eds.) Edinburgh University Press, Edinburgh, pp. 127-140.

Pohl, I. (1973) The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. *Proc. IJCAI 3*, pp. 12-17.

Pohl, I. (1975) Practical complexity and the design of algorithms. *Informatie* (Nederland) **17**, 205-211.

Robinson, J.A. (1965) A machine-oriented logic based on the resolution principle. *JACM* **12**, 23-41.

Rosenberg, R.S. and J. Kestner (1972), Look-ahead and one-person games. *Journal of Cybernetics* **2**, 4, 27-42.

Ross, R. (1973) Adaptive aspects of heuristic search. *Ph.D. Thesis*, University of Edinburgh, Edinburgh.

Shannon, C.E. (1950) Programming a computer for playing chess. *Phil. Mag.* **41**, 256-75.

Slagle, J., and P. Bursky (1968) Experiments with a multipurpose theorem-proving heuristic program. *JACM* **15**, 85-99.

Turing, A.M. (1953) Digital computers applied to games, *Faster than Thought*, (Bowden, ed.), London: Pitman, pp. 286-95.

4

Variable Range Restrictions in Resolution Theorem Proving

Sharon Sickel

University of California
Santa Cruz, California

The vast number of clauses generated before a proof is found is one of the central problems of classical theorem proving. Many clauses generated by random resolution or resolution restricted by traditional syntactic strategies cannot possibly be used in a proof because the terms that were substituted for the variables in generating the clauses are inconsistent with all possible proofs. Investigating substitutions for variables provides a new way of attacking the problem.

The essence of the approach discussed here is to determine which instances of the original clauses are required for a proof. Information is collected on variables that will specify the ranges of values that they may take on.

We also describe a measure of proof complexity. Proofs are attempted allowing increasing orders of complexity, and *all* proofs of a given order of complexity are found before the next order is attempted. The solution to a given problem is a set of proof schemas representing all proofs of the lowest complexity possible, rather than a single proof as with traditional methods.

INTRODUCTION

Automatic theorem proving has always been plagued by the derivations of theorems irrelevant to the desired one. After the grossly inefficient British Museum Algorithm (Feigenbaum and Feldman, 1963), the Resolution Principle (Robinson, 1965,1967,1968) seemed to renew hopes for an efficient procedure. For a theorem to be proved by the Resolution Principle, the axioms and the negation of the theorem were assumed to be simultaneously satisfiable. If a contradiction could be derived from that assumption, then the theorem was considered proved. The problem of generating any contradiction appeared easier than that of generating a particular formula. In addition, the Conjunctive Normal Form (CNF) (Mendelson, 1964) caused many equivalent statements to be represented by a single formula, so that the number of required axioms was reduced somewhat. However, the Resolution Principle approach is also severely restricted by the growth problem.

The Set of Support Strategy (Wos, *et al.*, 1965) requires that the first resolution in every branch of the proof tree involves a clause representing at least part of the negation of the theorem to be proved. This strategy increases the direction of proofs by requiring that each partial derivation relates in at least a limited way to the theorem to be proved. It fails, nevertheless, to make theorem proving practical for most applications.

Most other existing strategies attempt to reduce the combinatoric explosion of clauses by disallowing some choices for resolution and eliminating some already generated clauses. However, the deletion criteria are usually unrelated to the clauses' actual worth. As Shostak (Shostak, 1974) aptly characterized deletion strategies–they "prune garbage and gold in equal proportion."

In general, much effort is wasted on partial derivations that cannot lead to a proof and in developing many variations of the same proof simultaneously. The combinatorics quickly become overwhelming.

In proving theorems, people don't use axioms blindly and become involved in the combinatoric problems that are associated with machine proof techniques. For example, if the axiom:

All mammals are vertebrates,

is known, a human would generally not consider the instances of that axiom:

If a cat is a mammal, then a cat is a vertebrate.

and

If a car is a mammal, then a car is a vertebrate.

to be equally likely to be applicable to a proof even if the problem dealt with both cats and cars. The choice would depend upon the context of the problem. The first instance might be useful in proving that a cat is an animal–the second for proving that a car is not a mammal. (The latter may seem nonsensical; however, it is a fact that is probably not directly stored in most people's minds, but is easily deducible, given some knowledge of cars and mammals.)

Most instances of most axioms are irrelevant to any given problem. Some common sense is used to select the necessary instances of axioms, and the proposed approach helps to capture and formalize some of that sense. The useful instances can be determined by considering the interrelationships of the clauses.

A specific theorem proving example is:

Example 1

1. $\overline{L}(x,b)\ \overline{N}(y)\ \overline{L}(x,y)\ S(y,c(x))$
2. $\overline{N}(x)\ \overline{S}(x,y)\ \overline{S}(y,k(x))\ N(y)$
3. $S(x,k(x))$
4. $N(c(x))$
5. $C(k(x))$

6. S(x,x)
7. $\overline{S}(x,y)\ \overline{S}(y,x)\ E(y,x)$
8. L(x,k(c(x))
9. $\overline{E}(y,x)\ \overline{C}(y)\ C(x)$
10. L(a,b)
11. $\overline{C}(c(a))$

There are many possible unifications for this set of clauses, creating many possible instances of the clauses that can be generated by resolution. However, any possible proof from these clauses uses exactly the following instances:

1. $\overline{L}(a,b)\ \overline{N}(k(c(a)))\ \overline{L}(a,k(c(a)))\ S(k(c(a)), c(a))$
2. $\overline{N}(c(a))\ \overline{S}(c(a), k(c(a)))\ \overline{S}(k(c(a)), k(c(a)))\ N(k(c(a)))$
3. S(c(a), k(c(a)))
4. N(c(a))
5. C(k(c(a)))
6. ——— (none)
7. $\overline{S}(c(a), k(c(a)))\ \overline{S}(k(c(a)), c(a))\ E(k(c(a)), c(a))$
8. L(a,k(c(a))
9. $\overline{E}(k(c(a)), c(a))\ \overline{C}(k(c(a)))\ C(c(a))$
10. L(a,b)
11. $\overline{C}(C(a))$

The ability to determine the useful instances while searching for a proof would aid theorem provers substantially by saving resources that would be spent in creating dead-ended derivations.

The approach to be presented searches for the useful instances and in the process generates steps that make up a proof.

METHOD

Preliminary definitions

The object in resolution-based automatic theorem proving is to contradict the negation of the theorem, i.e., to resolve away all literals in an element of the set of support. In the process of resolving away a literal in the starting clause, other literals may be added. For example, given the starting clause AB, if we use clause $\overline{B}CD$ to resolve literal B away, literals C and D become part of the resolvent. We call literal $\overline{B}$ a *deleting literal* and literals C and D *residual literals* in the context of this resolution. To cause the net improvement of eliminating literal B from starting clause AB, we need to also eliminate literals C and D. We informally define a *partial solution* for a literal to be the sequence of resolutions that will eliminate that literal from the clause without adding other literals.*

*Work on Ordered Resolution and Model Elimination has dealt with the issue of eliminating one literal at a time (Gelperin, 1973; Kowalski and Kuehner, 1971; Loveland, forthcoming; Reiter, 1971), and some useful results are available.

A *total solution* for a clause consists of a partial solution for each literal in that clause. In the ground case, a total solution for a clause exists if and only if there exists one or more partial solutions for each literal. As we shall see in the next section, such is not true for the general case, where the partial solutions making up the complete solution must be collectively consistent in the way variables are instantiated.

Interrelationships of ranges of variables

The key to finding a proof in the predicate calculus lies in the ability to choose appropriate substitutions for variables such that a counterexample is created. The major problem here is that unification of two literals gives the most general substitution required without considering the context of those literals. Making a substitution into a literal (and therefore its entire clause) may instantiate a variable in another literal of that clause such that there is no way to derive a partial solution for it, but the dead-end may be invisible until several more resolution steps are made. Consider the following example, where s↔t symbolizes that the two terms, s and t, must be unified, but does not specify exactly how the substitution will be performed. We say that two literals *match* if they are unifiable.

Example

Suppose clause C is $(A(x,y,z) \vee \overline{B}(y))$. To use C in a proof, we must be able to resolve away both literals. Suppose that there are two matches for A(x,y,z) and three matches for $\overline{B}(y)$, and the unifications required for those matches are as follows.

$$A(x,y,z); \quad \begin{pmatrix} x \leftrightarrow b \\ y \leftrightarrow f(b) \\ z \leftrightarrow b \end{pmatrix}, \quad \begin{pmatrix} x \leftrightarrow g(a) \\ y \leftrightarrow a \\ z \leftrightarrow a \end{pmatrix}$$

$$\overline{B}(y): \quad (y \leftrightarrow b), \quad (y \leftrightarrow f(v)), \quad (y \leftrightarrow g(a))$$

Since "y" in both literals represents the same variable, any substitution used for the match of one literal must be consistent with at least one substitution for the other literal. Otherwise, we could derive clauses containing literals that have no matches, and we would have expended resources in reaching dead ends. For instance, resolving A(x,y,z) against $\overline{A}(g(a),a,a)$ would produce a resolvent containing $\overline{B}(a)$. By looking at the matches of $\overline{B}(y)$ we can see that $\overline{B}(a)$ will have no matches, and this new resolvent cannot possibly be used in a proof. In this example, only the substitution [f(b)/y] could possibly be used for both literals. Therefore, only the first match for A(x,y,z) and the second match for $\overline{B}(y)$ should *ever* be used. The other matches may be eliminated since they produce inconsistencies.

At this first level, it is easy to require that any substitution applied to a given

literal is consistent with at least one substitution needed to unify each of the other literals. As the ranges of some variables are restricted, other variables depending on them will also be restricted.

Checking to see that each unification of a literal L is consistent with at least one unification for every other literal in the clause containing L is only the beginning. As we step-by-step construct partial solutions and progressively specify the substitutions, we check to see that for each partial solution being developed there are consistent partial solutions being developed for the other literals in the clause. Although there may be many ways to construct partial solutions for literals in a clause, the number of consistent combinations, i.e., total solutions, may be few.

For example, given the starting clause (A(x,y) B(x) C(y)), suppose that the substitutions associated with each of the partial solutions for literals in this clause are:

[a/x,b/y], [f(z)/x,c/y], or [g(h(b))/x,f(g(a))/y] for A(x,y)
[f(a)/x] or [g(h(b))/x] for B(x)

and

[b/y], [c/y], or [g(h(b))/y] for C(y).

The only substitution consistent with a partial solution for each of the three literals in the starting clause is [f(a)/x,c/y]. Therefore, if these are the only existing partial solutions, then the only relevant instance of the given clause is A(f(a), c) B(f(a)) C(c)), and there is only one total solution.

Problem representation

Clauses will be manipulated differently in this method than in traditional approaches, and the structure of the representation should be appropriate for the operations on the structure. For that reason, each problem is represented by a *Clause Interconnectivity Graph*†, defined as follows:

1) One node exists in the graph for each literal of each clause.††
2) The nodes are partitioned by clause membership, i.e., two literals in the same clause will have corresponding nodes in the same partition.
3) Edges exist that connect any two literals that are unifiable. Each edge is labeled with a most general unifying substitution that unifies

---

†This representation, while developed independently, is very similar to Kowalski's Connection Graphs (Kowalski, 1974). Here, partitions of nodes are required to show clause structure since the original clauses are not retained. See ††. The operations defined on Clause Interconnectivity Graphs are very different from the operations on Connection Graphs.

††The names of the predicates and terms could be eliminated. No new clauses are generated; therefore the names are needed only while performing the unifications on the input set and for interaction with the user. If the implementor of a system prefers, they can be easily maintained external to this graph.

the two literals. This means that all unifications that exist between literals in the input set of clauses will have to be found before the search for a proof begins. However, the total number of unifications are few since 1) the input set is generally small, and 2) this is the only stage at which unification will be performed because we will generate no new clauses in search for a proof.

See Figure 1.

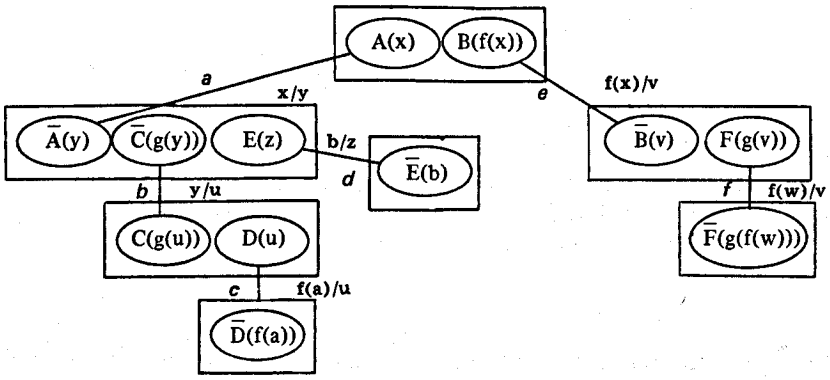
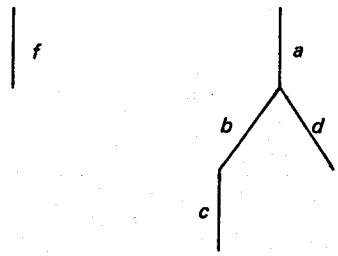


FIG. 1. Clause interconnectivity graph for the set of clauses: $\{A(x)B(f(x)), \bar{A}(y)\bar{C}(g(y))E(z), \bar{F}(g(f(w))), \bar{B}(v)F(g(v)), \bar{D}(f(a)), \bar{E}(b), C(g(u))D(u)\}$. (In the figure nodes are circled, partitions are boxed, and edges have been given names *a–f*.) In this and some other clause interconnectivity graphs appearing in this paper, the predicates and terms are left in only for purposes of discussion. They are not used by the theorem proving procedure.

In the context of a clause interconnectivity graph, the analog to resolving two literals is to "walk" across the path linking the two. The literal that we are walking toward on the path is the *deleting literal* (It deletes the literal we are walking from). The other literals in the clause partition containing that deleting literal are the *residual literals*.

PS(L) denotes a *partial solution* for literal L in a clause interconnectivity graph and is defined as a tree that represents walks that could be taken to eliminate the given literal and all residual literals that are produced. For example, the partial solutions for literals F(g(v)) and A(x) from Figure 1, respectively, are:



A partial solution for a node, N, can be any subgraph of a clause interconnectivity graph that forms a tree having N as its root. For this example, the tree structure is obvious; for other problems such will not always be the case. This issue will be dealt with in detail below.

TS(C) denotes a *total solution* for a clause C, (where $C = C_1 C_2 \ldots C_n$, each C_i a literal), and is defined as a set $\{X_1, X_2 \ldots X_n \mid X_i \in PS(C_i)\}$.

A *refutation* of a set of clauses is TS (Start Clause) where Start Clause is assumed to be a member of the Set of Support.

Search

Blind search

A *blind search* could be defined that starts with a set of *active nodes*; that set consists of the nodes representing literals in one clause of the set of support. A walk is allowed from any active node along an existing edge to any other node. The node walked from is deleted from the active set. The residual literals created by the walk are added to the set of active nodes. The active nodes represent the literals that have been introduced into the refutation, but have not yet been eliminated. Therefore, a proof has been found when all introduced literals are eliminated, i.e., when the active set is empty. Variables in residual literals must be standardized apart from the variables already in the active set. Substitutions required by walking an edge should be made to the entire active set, to guarantee consistency of substitutions. When a partition is re-entered, we must consider this a new instance of the partition. (This is taken care of by standardizing apart the variables in residual literals.) There needs to exist an appropriate backtracking algorithm to recover when the search has been blocked by having no edges whose substitutions are consistent with those already made into the active set.

The blind search, as described, is equivalent to a pure depth-first search for the input strategy of resolution. The method is neither complete, nor very effective. It was presented only to give the reader some familiarity with the graph representation and the analogy with resolution.

Graph unrolling

The clause interconnectivity graph shown in Figure 1 has two very convenient properties. The first is that each node has only one edge attached to it, so that there is never any choice to be made. The advantage to this is obvious. The second advantage is that the graph is actually a tree. The partial solution of each node is exactly the subtree whose root is that node. If a clause interconnectivity graph does not have the property of being a tree, then the subgraph that represents a partial solution can be conceptually infinite, by being defined in terms of itself. For this reason, we define the notion of complexity, and we use that definition to divide the search into an enumeration of workable chunks.

A partial solution is defined as containing a *loop* if any path from root to leaf

of the partial solution contains more than one occurrence of any node of a given partition. In terms of walks, a loop is a path that returns to a partition visited previously in that walk.

For example, consider the following clause interconnectivity graph.

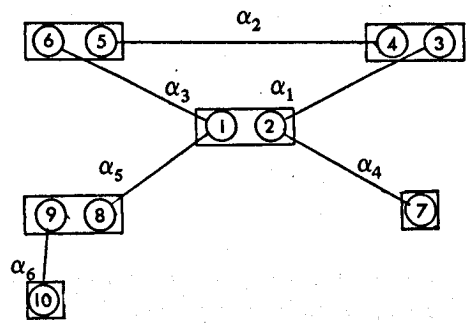


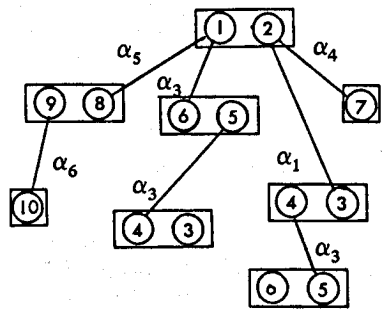
FIG. 2

The partial solutions are:

$$PS(1) = \{\alpha_5\alpha_6, \alpha_3\alpha_2\alpha_1\ PS(1)\}$$
$$PS(2) = \{\alpha_4, \alpha_1\alpha_2\alpha_3\ PS(2)\}.$$

Each of the above partial solutions are defined in terms of themselves. The way we handle this is to define an enumeration of trees that represent the loops in the graph being further and further unrolled. At the k-th step of the enumeration, we allow loops to be unrolled k times. For the graph in Figure 2, using [① ②] as the start clause, the enumeration produces:

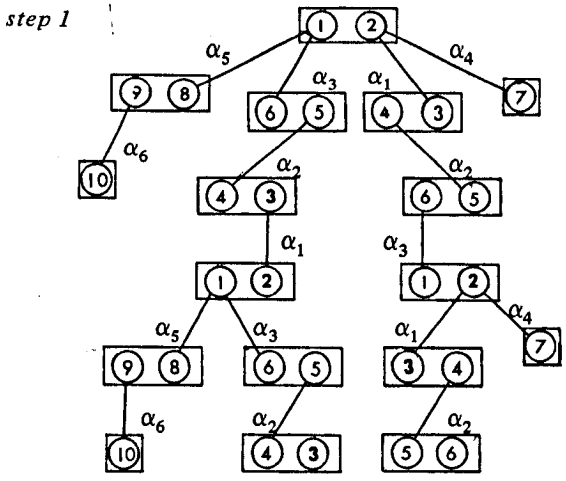
step 0



No more expansion would be allowed without including a loop.

$$PS(1) = \{\alpha_5\alpha_6\}$$
$$PS(2) = \{\alpha_4\}$$

If $(\alpha_5\alpha_6)$ and (α_4) are consistent, then we are finished. The tree represents a proof.



Loops were unrolled once.

$$PS(1) = \{\alpha_5\alpha_6,\ \alpha_3\alpha_2\alpha_1\alpha_5\alpha_6\}$$

$$PS(2) = \{\alpha_4,\ \alpha_1\alpha_2\alpha_3\alpha_4\}$$

If elements from each of PS(1) and PS(2) are consistent, then we are finished. Otherwise generate step2, etc.

As we have defined our process so far, we still have the problem that we construct only input proofs. In order to overcome this limitation, we allow another kind of walk. We define a *merge loop* to be a loop such that:

(i) the node at each end of the loop is the same (as opposed to being merely in the same partition).

(ii) the substitutions used are collectively consistent even if no new variables are added when re-entering the node at the loop ends.

In Figure 3, path abd is a merge loop. The node representing literal A(x) is the node appearing at both ends of the loop. Substitution [f(z)/x, f(z)/y, f(z)/u] is consistent with every edge in the loop and requires only one instance of x, thus satisfying condition (ii) of the merge loop definition.

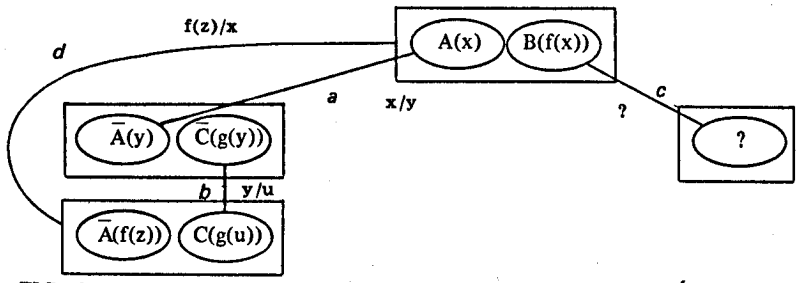
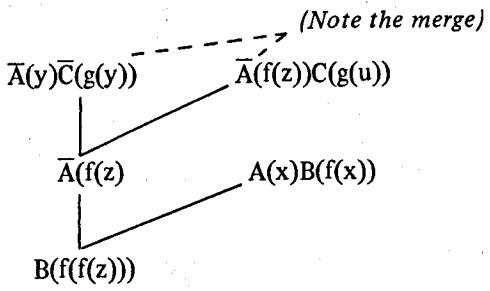


FIG. 3. Clause interconnectivity graph for the set of clauses: $\{A(x)B(f(x)), \bar{A}(y)C(g(y)), \bar{A}(f(z))C(g(u))\}$.

If we impose the restriction that a single occurrence of the end node is used, then a merge loop indicates that two other nodes are being eliminated by a single occurrence of the end node, i.e. the two nodes matched by the end node must first be merged. The resolution deduction representing merge loop abd is:



In the process of unrolling the graph, we will allow a merge loop to be used without counting as a loop; for example, a merge loop is allowed even in step 0. If however, a proof is not found in the current step, a regular loop must be added at the next step. Figure 4 shows the search trees at step 0 and step 1 for the graph in Figure 3.

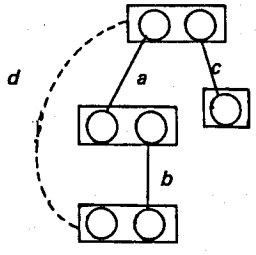


Fig. 4a

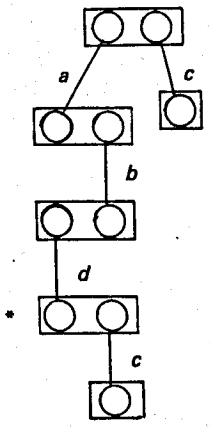


Fig. 4b

FIG. 4. Search trees for the example in Fig. 3. Either of the Figures 4a or 4b could be required, depending on the unknown node. **FIG. 4a.** Step 0 search tree for Fig. 3. Dotted line indicates a merge loop. **FIG. 4b.** Step 1 search tree for Fig. 3. If the merge loop *abd* of Fig. 4a is inconsistent with *c* then we need to proceed with this step. Note that a *new instance* of the variable x will be added at *. In this case the substitution for x on path *a* need not be consistent with the substitution for x on path *d*.

Given the merge-loop exception, we can now define *proof complexity*. The proof of a theorem is of complexity n if at least one partial solution used in the proof contains n non-merge loops, and no partial solution contains more than n non-merge loops. Therefore, for a given complexity level, all partial solutions of literals are finite, and we can represent them as trees instead of graphs.

For example, the following proofs are of complexities 0, 1, and 2 respectively.

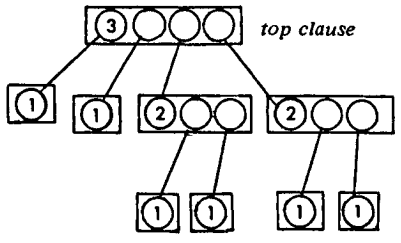


FIG. 5a. Complexity 0

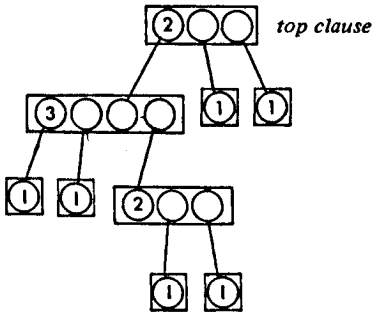


FIG. 5b. Complexity 1

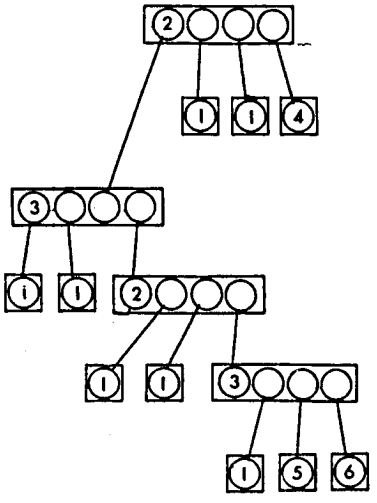


FIG. 5c. Complexity 2

FIG. 5. Examples of proofs of complexities 0,1,2. Partition numbers are indicated so loops will be recognizable to the reader.

Notice that Figures 5a and 5b are two different refutations for the same graph. The choice of the starting clause can affect the minimum complexity in which a proof can be found.

One may wonder what sort of problems require proofs of complexity greater than zero or what higher complexity proofs mean in intuitive terms. A proof of complexity greater than zero applies an axiom to itself. For instance, you may derive the special form of associativity:

$$(a \cdot (c \cdot d)) \cdot e = a \cdot (c \cdot (d \cdot e)$$

by applying associativity to itself thus:

$$(a \cdot (c \cdot d)) \cdot e \Rightarrow a \cdot ((c \cdot d) \cdot e) \Rightarrow a \cdot (c \cdot (d \cdot e)).$$

Search procedure

The search procedure then becomes:

1. $k \leftarrow 0$.
2. Generate tree for complexity k.
3. Does there exist a total solution for the start clause. If yes, terminate. If no, go to 4.
4. $k \leftarrow k+1$.
5. Go to 2.

The potential for exponential growth may seem to be as great as with traditional resolution based systems. However, two restrictions on substitutions are imposed:

i) every path from root to leaf must be substitution consistent.
ii) every partial solution for node N being constructed, whether complete or not, must be substitution consistent with at least one partial solution for every other node in N's partition.

By requring these two consistency restrictions we cause some branches of the search tree to be pruned away. As the tree gets deeper, the constraints on the substitutions increase, and the branching decreases. In this way, as the search progresses, we can use information we deduced about ranges of variables to improve the direction of search.

Proof schemata

If a total solution is found, then the steps making up the partial solutions in the total solution describe the resolutions needed for a proof. Performing the resolutions in any order will produce a proof, as long as the merge loops are handled properly. Therefore, a total solution is really a proof schema representing proofs that can be made from the specified steps.

BIBLIOGRAPHY

Anderson, R., and W.W. Bledsoe (1970) A linear format for resolution with merging and a new technique for establishing completeness. *JACM* 17, 3, 525-354.

Andrews, Peter B. (1968) Resolution with merging, *JACM* 15 3, 367-381.

Boyer, R.S. and J S. Moore (1972) The sharing of structure in theorem-proving programs. *Machine Intelligence* 7. (eds. B. Meltzer and D. Michie) John Wiley & Sons, New York.

Feigenbaum, E. and J. Feldman (1963) (Editors) *Computers and Thought*. McGraw-Hill, New York, 1963.

Gelperin, David (1973) Deletion-directed search in resolution-based proof procedures. *Proc IJCAI3* 47-50.

Kowalski, R.A. and D.G. Kuehner (1971) Linear resolution with selection function, *Artificial Intelligence*, 227-260.

Kowalski, R. (1974) Logic for problem solving. Dept. of Computational Logic, Memo No. 75, University of Edinburgh.

Loveland, D.W. and M.E. Stickel (1973) A hole in goal trees: Some guidance from resolution theory: *Proc IJCAI3*, 153-161.

Loveland, D.W. (forthcoming) *Mechanical Theorem Proving: Resolution Theory* (tentative title), North Holland.

Luckham, D. (1969) Refinement theorems in resolution theory. *Stanford A.I. Project Memo 81*, Stanford University, Stanford, CA, (March 1969).

Luckham, D. (1968) Some tree-paring strategies for theorem proving. *Machine Intelligence* 3, Oliver and Boyd, Edinburgh, 95-112.

Mendelson, E. (1964) *Introduction to Mathematical Logic.* Van Nostrand-Reinhold, Princeton, New Jersey.

Reboh, Rene, Bertram Raphael, Robert A. Yates, Robert E. Kling, Carlos Velard (1972) Study of automatic theorem-proving programs. Stanford Research Institute.

Reiter, Raymond (1969) The predicate elimination strategy in theorem proving. Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada.

Reiter, Raymond (1971) Two results on ordering for-resolution with merging and linear format. *JACM* 18, 630-646.

Robinson, J.A. (1965) A machine-oriented logic based on the resolution principle. *JACM* 12, 1, 23-41.

Robinson, J.A. (1967) A review of automatic theorem-proving. *Proc. Symposia in Applied Mathematics*, Amer. Math. Soc., Providence, R.I., 1-18.

Robinson, J.A. (1968) The generalized resolution principle, *Machine Intelligence* 3, (ed. D. Michie), Edinburgh University Press, Edinburgh, 77-93.

Shostak, Robert E. (1974) Refutation graphs and resolution theorem proving, Harvard University.

Sickel, Sharon (1974) Growth limitations for automatic theorem proving. Board of Studies in Information Sciences, University of California, Santa Cruz, California.

Slagle, James R. (1974) Automated theorem-proving for theories with simplifiers, commutavity, and associativity. *JACM* 21, 4, 622-642.

Slagle, J.R. (1967) Automatic theorem proving with renamable and semantic resolution. *JACM* 14, 4, 687-697.

Wos, L., D.F. Carson, and G.A. Robinson (1964) The unit preference strategy in theorem proving. *Proc AFIPS FJCC*, 26 Pt. 1, Spartan Books, Washington, D.C., 615-621.

Wos, L., G.A. Robinson, and D.F. Carson (1965) Efficiency and completeness of the Set of Support strategy in theorem proving. *JACM* 12, 4, 536-541.

Yates, R., B. Raphael, and P. Hart (1970) Resolution graphs. *Artificial Intelligence*, 257-289.

5

A Formal Model for Space-filling Puzzles

W. M. McKeeman

University of California
Santa Cruz, California

INTRODUCTION

The Soma Cube, Instant Insanity, and Eight Queens are all examples of space filling puzzles. As concrete puzzles they are challenging because the number of ways of attempting a solution is exponentially larger than the number of actual solutions. The same property makes them interesting computationally.

In geometry, a space filling is a regular pattern of objects that fit together to fill up the entire space in which they are imbedded. An infinite chessboard, for example, uses squares to fill two-space. If the chessboard squares are replaced by octagons, smaller squares rotated 45° can be inserted in the gaps, hence octagons and small squares can also be used to fill two-space (Fig. 1). There are many such examples.

Suppose we removed a finite number of finite regions from a space filling. The removed regions, called pieces, each containing one or more of the basic underlying objects of the space filling, can be quite irregular in shape. The space-filling puzzle consists of putting them back without knowing how they came out in the first place.

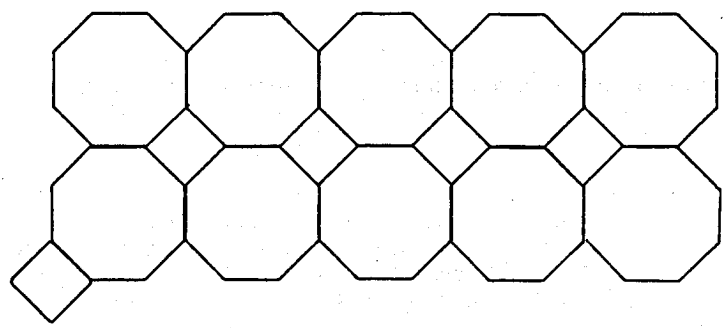


FIG. 1. A space filling

The first objective of this paper is the presentation of a formal model, using sets and groups, in which all of the above can be represented. The second objective is to catalog a number of specific examples. Proofs of the validity of the model are given in *Efficient Solution of Space-filling Puzzles* (McKeeman, Fay & Pennello, 1976).

THE FORMAL MODEL

Let s be the set of all objects in the space filling. Let P_0 be the set of pieces removed to define the puzzle. Each member of P_0 is a subset of s, and they are mutually disjoint. Let h be the union over P_0. Then h is the hole in s left by removing the pieces.

Due to the regularity of the space filling, each piece in P_0 can be moved to other positions in s. Because the moves are reversible and associative, they can be represented by a permutation group, G, on s. All possible piece positions are then in $\cup P$† where the set P is given by

$$P = \{G(p) \mid p \in P_0\}$$

and

$$G(p) = \{g(p) \mid g \in G\}$$

and

$$g(p) = \{g(x) \mid x \in p\} .$$

Note that P is unaffected by substituting an arbitrary transformation g(p) for each original p in P_0. Thus we may generalize P_0 to any set of pieces that can be *transformed* into a solution set.

Whatever h is picked as a puzzle definition, there are other equally good choices that could have been made. The set

$$G(h) = \{g(h) \mid g \in G\}$$

gives the set of all holes equivalent to h, any one of which is a suitable target. That is to say, we still regard the Soma Cube as solved whether Donald Michie does it sitting at his desk in Edinburgh or Bill McKeeman does it standing on his head in Santa Cruz.

A solution is a subset r, of $\cup P$, such that $\cup r \in G(h)$ and the pieces in r are mutually disjoint. That is, they fill some hole equivalent to h and do not overlap. The set of all solutions, R, is given by

$$R = \{r \mid r \subseteq \cup P, \cup r \in G(h), \text{disjoint}(r)\}$$

If one such solution can be moved intact to another solution via G, the solutions are equivalent. Thus the classes of *unique* solutions to the puzzles are given by

$$S = \{G(r) \mid r \in R\}$$

---

†For any set of sets X, $\cup X \stackrel{\text{def}}{=} \cup_{x \in X} x$, the union over its members.

where, as before,

$$G(r) = \{g(r) \mid g \in G\}$$

and

$$g(r) = \{g(p) \mid p \in r\}$$

and so forth.

To summarize,

Given s, P_0 h and G, solve for

$P = \{G(p) \mid p \in P_0\}$,
$R = \{r \mid r \subseteq \cup P, \cup r \in G(h), \text{disjoint}(r)\}$
$S = \{G(r) \mid r \in R\}$

The above is not (at least not transparently) an algorithm since all of s, G, R and the elements of S may be infinite.

THE SOMA CUBE

The basic object out of which the Soma Cube is constructed is a unit cube. There are seven pieces, each built by gluing several cubes together by their faces. They can be characterized as having at most four unit cubes, and each piece having an inside (concave) corner (Fig. 2). The finished puzzle is a 3x3x3 cube.

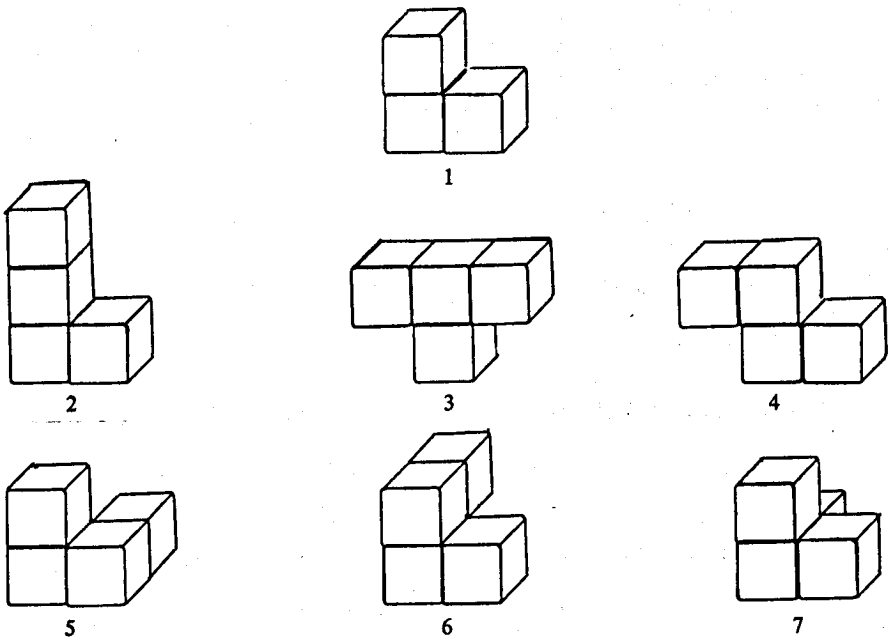


FIG. 2. Pieces of the Soma Cube

To map the puzzle onto the formal notation we will first construct an infinite Cartesian space filled with unit cubes. A particular unit cube can be specified by the coordinate of its center. Let I be the integers, and the set of triples I^3 be the centers of the unit cubes.

To characterize a Soma Cube piece we must give both an example of a subspace that it fills and a name. The integers 1 to 7 (Fig. 2) can serve as names. Thus s must contain I^3 and the integers 1 to 7.

$$s = I^3 \cup \{1,2,3,4,5,6,7\}.$$

The initial piece descriptions are subsets of s given by

$$\begin{aligned} P_0 = \{&\{000, 100, 010, 1\}, \\ &\{000, 100, 200, 010, 2\}, \\ &\{000, 100, 200, 110, 3\}, \\ &\{000, 100, 110, 210, 4\}, \\ &\{000, 100, 010, 101, 5\}, \\ &\{000, 100, 010, 011, 6\}, \\ &\{000, 100, 010, 001, 7\}\}. \end{aligned}$$

The region defining the completed puzzle is given by

$$\begin{aligned} h = \{&000, 001, 002, 010, 011, 012, 020, 021, 022, \\ &100, 101, 102, 110, 111, 112, 120, 121, 122, \\ &200, 201, 202, 210, 211, 212, 220, 221, 222, \\ &1, 2, 3, 4, 5, 6, 7\}. \end{aligned}$$

The apparently redundant appearance of the piece names in the descriptions is a consequence of the definition of R. In taking subsets of $\cup P$, it is possible to get several piece positions generated from the same starting member of P_0. For the Soma Cube this is not to be allowed since each piece may be used only once. For the Eight Queens problem it *is* desired since there is only one piece and it is used eight times. The (occasional) addition of the piece names in puzzle descriptions is simply a way to increase the generality of the formalization. It can be thought of in another way which makes it seem more natural. The Soma Cube is considered solved when each of 34 conditions is met exactly once: 27 different unit cubes must be filled and 7 different pieces must be used. As expected, h for the Soma Cube therefore is of size 34.

The group G is generated by three translations:

$$\begin{aligned} g_1([i,j,k]) &= [i+1,j,k], \; g_1(n) = n \\ g_2([i,j,k]) &= [i,j+1,k], \; g_2(n) = n \\ g_3([i,j,k]) &= [i,j,k+1], \; g_3(n) = n \end{aligned}$$

and two rotations:

$$\begin{aligned} g_4([i,j,k]) &= [-j,i,k], \; g_4(n) = n \\ g_5([i,j,k]) &= [i,-k,j], \; g_5(n) = n \end{aligned}$$

and a reflection:

$$g_6([i,j,k]) = [j,i,k],\ g_6(n) = n,$$
$$\text{except } g_6(5) = 6 \text{ and } g_6(6) = 5.$$

This completes the description of the Soma Cube as a space-filling puzzle. To describe any other puzzle that can be built out of the Soma Cube pieces, one need only make the corresponding change in h.

INSTANT INSANITY

The puzzle consists of four colored cubes. The solution is achieved when the cubes are aligned (say stacked up) so that each of the four colors appears on each side of the stack. The sides that happen to be faced up or down are ignored. The basic objects out of which one builds Instant Insanity are color/direction pairs and piece names. Let the colors be $\{r,w,b,g\}$ (meaning red, white, blue, green), and the four directions be $\{N,E,S,W\}$, and the piece names be $\{1,2,3,4\}$. Then s is given by

$$s = \{rN, rE, rS, rW, wN, wE, wS, wW,$$
$$bN, bE, bS, bW, gN, gE, gS, gW,$$
$$1, 2, 3, 4\}$$

Each original colored cube has three initial positions in P_0 corresponding to each opposite pair of faces in the ignored (vertical) orientation. Unfortunately the published literature on Instant Insanity and some commercial versions of the puzzle do not have the same coloring. The one given here is taken from Brown's paper.

$$P_0 = \{\{rN,bE,rS,gW,1\},\ \{bN,bE,wS,gW,1\},\ \{rN,wE,rS,bW,1\},$$
$$\{gN,wE,bS,rW,2\},\ \{gN,wE,wS,rW,2\},\ \{gN,wE,bS,gW,2\},$$
$$\{gN,wE,rS,bW,3\},\ \{rN,wE,wS,bW,3\},\ \{gN,wE,rS,rW,3\},$$
$$\{gN,wE,gS,bW,4\},\ \{rN,wE,gS,bW,4\},\ \{gN,gE,gS,rW,4\}\}.$$

The solution space h is identical to s. The group G is given by a rotation mapping N to E to S to W to N, and a reflection mapping N to S to N.

EIGHT QUEENS

From the viewpoint of an attacking queen, the world consists of a row, column, and two oppositely sloping diagonals. The Eight Queens problem is solved when all eight are on the chessboard and their views do not overlap. There are several statements of the problem, each of which leads to a different result, basically because of how one chooses to view symmetry. There are 92 solutions, 12 unique under rotation and reflection, and only 6 unique under rotation, reflection and end-around translation. A formalization leading to 6 solutions is

given here. On an infinite board there are an infinite number of rows, columns and diagonals of both types. Each can be uniquely represented by an integer.

Letting I stand for the integers, s is given by

$$s = \{r_i \mid i \in I\} \cup \{c_i \mid i \in I\} \cup \{p_i \mid i \in I\} \cup \{n_i \mid i \in I\}$$

where r, c, p and n stand for row, column positive diagonal and negative diagonal respectively.

An initial queen position is given by $\{r_0, c_0, p_0, n_0\}$. After the queens are properly placed all the rows and columns will be occupied but some diagonals will remain unoccupied. Thus we must add in one "free" diagonal to fill up the remaining space, e.g., $\{p_0\}$. Thus P_0 is given by

$$P_0 = \left\{\{r_0, c_0, p_0, n_0\}, \{p_0\}\right\}$$

The solution space h is given by restricting the range of r_i and c_i to $0 \leqslant i \leqslant 7$, the range of p_i to $-7 \leqslant i \leqslant 7$, and the range of n_i to $0 \leqslant i \leqslant 15$. The group G is generated by two translations, a rotation and a reflection. The first translation maps i to i+1 for each of c_i, p_i and n_i, but leaves r_i fixed. The second translation maps i to i+1 for r_i and p_i, i to i-1 for n_i, and leaves c_i fixed. The rotation maps r_i to c_{-i}, c_i to r_i, p_i to n_i, n_i to p_{-i}. The reflection leaves r_i fixed, maps c_i to c_{-i}, p_i to n_{-i}, and n_i to p_{-i}.

N^M ROOKS

A hyperrook operates on an M+1 dimensional hyperchessboard of side N. The view of the attacking hyperrook is M+1 lines parallel to the edges of the hyperchessboard. The N^M rooks problem is solved when they are all on the hyperchessboard and their views do not mutually overlap. As in the Eight Queens, there are several formalizations of the problem each due to a different view of symmetry. Some special cases have been solved in closed form (Larson, 1974).

Suppose N is odd. Then let

$$J = \left\{i \mid i \in I, |i| \leqslant (N-1)/2\right\}$$
$$s = h = J^{M+1}$$

For an element $j \in s$, j_i is the i^{th} component

$$f_i = \left\{j \mid j \in s, k \neq i \text{ implies } j_k = 0 \text{ for } 0 \leqslant k \leqslant M\right\}$$

is the i^{th} file attacked from the center of the board. The initial piece position is given by

$$P_0 = \left\{\cup\left\{f_i \mid 0 \leqslant i \leqslant M\right\}\right\}.$$

The group G is generated by M+1 translations, M rotations and a reflection. The translations are end-around. That is, if some $j_i = (N-1)/2$, then a translation in

the i^{th} direction will take it to -(N-1)/2. The rotations and reflections are defined as in the Soma Cube description.

A similar approach can be taken for N even. There are N^{M+1} cubes in s; there are $\frac{d}{dN} N^{M+1} = (M+1)N^M$ files. The problem can also be stated in terms of a set s of all files.

A FINITE MODEL

The solution space h may have some inherent symmetry under G. It is characterized by the set

$$G_h = \{g \mid g \in G, g(h) = h\}.$$

G_h is, in fact, a finite subgroup of G. It can be used to state an equivalent, finite, solution to the space filling problem.

Given s, P_0, h and G, solve for

$$P' = \{Q'(p) \mid p \in P_0\}$$

where

$$Q'(p) = \{p \mid p \in G(p), p \subseteq h\}$$
$$R' = \{r \mid r \subseteq \cup P', \cup r = h, \text{disjoint}(r)\}$$
$$S' = \{G_h(r) \mid r \in R'\}$$

All of P', R' and S' are finite because h is finite. The combinatorics are still overwhelming. For the Soma Cube, $\cup P'$ contains 688 piece positions, hence there are potentially 2^{688} subsets r to be examined. Nevertheless one can proceed to an implementable algorithm as follows.

Give names to the elements of h. Pick one more special name (signifying "outside h"), say "?", and define

$$h' = h \cup \{?\}.$$

Specify the initial piece positions in P_0 in terms of the new names in h. Let $G' = \{g \mid g \in G, p \in P_0 \text{ such that } g(p) \subseteq h\}$. Specify the transformations in G' as finite functions g' on h', always mapping "?" onto "?"; for other elements x of h', $g'(x) =$ "?" if the corresponding transformation g takes x outside of h, and $g'(x) = g(x)$ otherwise.

SUMMARY

Four space-filling puzzles were presented in terms of a model based on sets and permutation groups. Further, a general method was given for producing an equivalent finite formalization. The resulting algorithm is combinatorically explosive. It is shown elsewhere (McKeeman, Fay and Pennello, *loc. cit.*) how

the model can be used to reduce the combinatorics.

ACKNOWLEDGMENTS

A large number of people have made suggestions, and worked with the author on this problem. Cleve Moler suggested the algorithm implied in the definition of the set R in 1962. Todd Hirozowa implemented the first program at Santa Cruz to utilize the finite model and some of the combinatoric reductions mentioned above. Conversations with Ken Friedenbach led to the formalization in terms of G. Donald Knuth, Tom Atwater, Jim Horning, Donald Michie and Martin Gardner were very helpful in commenting on earlier drafts of this material and pointing out important references. Mike Fay, Tom Pennello and the author all benefited greatly from this help while writing the more definitive work noted above. This report is extracted from it. Many others, most especially the graduate students at Santa Cruz, have contributed in ways that were less specific but no less important. I thank them all.

REFERENCES AND OTHER RELEVANT PAPERS

Atwater, T. (Ed.) *Soma Addict*, 1, No. 1 & 2; 2, No. 1 & 2, (Parker Bros., Inc., Salem, MA (1970).

Brown, T.A. (1968) A Note on "Instant Insanity". *Mathematics Magazine*, p. 167.

DeLong, R. (1974) Solutions to the Soma Cube Problem. *SIGPLAN Notices*, p. 7 (October 1974).

Fletcher, J. (1965) A program to solve the pentomino problem by recursive uses of macros. *CACM*, 8, 10 p. 621.

Gardner, M., Mathematical Games, *Scientific American* (Sept. '72, Oct. '72, Oct. '73 and others).

Gardner, M. (1961) The Soma Cube, in *Mathematical Puzzles and Recreations*, chapter 6, (Simon and Schuster).

Hollander, D.H. (1963) An Unexpected Two-dimensional Space-group Containing Seven of the Twelve Basic Solutions to the Eight Queens Problem. *J. Rec. Math.* 6, 4, p. 287.

Kawai, S., Noshita, K., Takeuchi, I. (1975) On Backtrack Programming and Some Results on Combinatoric Puzzles, *Proc. Second USA-Japan Computer Conference*, p. 300.

Knuth, D.E., Estimating the Running Time of Backtrack Programs, in *The Art of Computer Programming*, Vol. 4 (Addison Wesley (to appear).

Larson, L.C. (1974) Essentially Different Nonattacking Rook Arrangements. *J. Rec. Math.* 7, 3, p. 178.

McKeeman, W.M. (1974) Solving Space-filling Puzzles, *CEP Report* 6, 1, Computer Evolution Project, The University of California at Santa Cruz, CA 95064.

McKeeman, W.M., Fay, M.J., Pennello, T.J. (1976) Efficient Solution of Space-filling Puzzles (to be published, 1976).

Meeus, J. (1973) Some Polyomino and Polyamond Problems, *J. Rec. Math.* 6, 3 p. 215.

Meeus, J. (1973) Tetracubes, *J. Rec. Math.*, 6 4, p. 257.

Torbijn, P.J. (1974) The Unknown World of Octaimonds, *J. Rec. Math* 7, 1, p. 1.

Wagner, N.R. (1973) Constructions with Pentacubes-2, *J. Rec. Math* 6 3, p. 211.

Wagner, N.R. (1972) Construction with Pentacubes, *J. Rec. Math.* 5, No. 4 p. 266.

Whinihan, M., and Trigg, C. (1973) Parity and Centerness Applied to the Soma Cube, *J. Rec. Math.* 6 No. 1 p. 61.

6

Achieving Several Goals Simultaneously*

Richard Waldinger

Artificial Intelligence Center
Stanford Research Institute

In the synthesis of a plan or computer program, the problem of achieving several goals simultaneously presents special difficulties, since a plan to achieve one goal may interfere with attaining the others. This paper develops the following strategy: to achieve two goals simultaneously, develop a plan to achieve one of them and then modify that plan to achieve the second as well. A systematic program modification technique is presented to support this strategy. The technique requires the introduction of a special "skeleton model" to represent a changing world that can accommodate modifications in the plan. This skeleton model also provides a novel approach to the "frame problem."

The strategy is illustrated by its application to three examples. Two examples involve synthesizing the following programs: interchanging the values of two variables and sorting three variables. The third entails formulating tricky blocks-world plans. The strategy has been implemented in a simple QLISP program.

It is argued that skeleton modelling is valuable as a planning technique apart from its use in plan modification, particularly because it facilitates the representation of "influential actions" whose effects may be far reaching.

The second part of the paper is a critical survey of contemporary planning literature, which compares our approach with other techniques for facing the same problems. The following is the outline of contents.

CONTENTS

0. INTRODUCTION

1. SIMULTANEOUS GOALS, PROGRAM MODIFICATION, AND THE REPRESENTATION OF ACTIONS

   1.1 A description of our approach
      1.1.1 Achieving primitive goals
      1.1.2 Goal regression
      1.1.3 Plan modification

*The research reported herein was sponsored by the National Science Foundation under Grant GJ-36146.

1.1.4 Protection
1.1.5 A very simple example
1.1.6 Skeleton models

1.2 Interchanging the values of two variables
1.2.1 Relations that refer to variables
1.2.2 The Solution to the two-variable problem

1.3 Sorting three variables
1.3.1 Sorting two variables
1.3.2 Achieving an implication
1.3.3 The solution to the three-sort problem

1.4 The Sussman "anomaly"

1.5 Limitations and next steps

2. THE REPRESENTATION OF ACTIONS AND SITUATIONS IN CONTEMPORARY PROBLEM SOLVING

2.1 The classical problem solvers
2.2 Regression and STRIPS operators
2.3 The use of contexts to represent a changing world
2.4 Influential actions
2.5 Escaping from the STRIPS assumption
2.6 The use of contexts to implement skeleton models
2.7 Hypothetical worlds
2.8 Complexity
2.9 Recapitulation

ACKNOWLEDGMENTS

REFERENCES

INTRODUCTION

My feet want to dance in the sun
My head wants to rest in the shade
The Lord says "Go out and have fun!"
But the landlord says "Your rent ain't paid!"

E.Y. Harburg, *Finian's Rainbow*

It is often easier to achieve either of two goals than to achieve both at the same time. In the course of achieving the second goal we may undo the effects of achieving the first. Terry Winograd points out in a *Psychology Today* article (Winograd, 1974) that his blocks program

> cannot carry out the command, "Build a stack without touching any pyramids," because it has no way to work on one goal (building a stack) while keeping track of another one (avoiding contact with pyramids).

The reasoning subprograms of his natural language processor "have a sort of one-track mind unsuited to complicated tasks."

In program synthesis, such "simultaneous goal" problems are rampant. A

typical example: the goal of a sort program is to rearrange an array in ascending order while ensuring at the same time that the resulting array is a permutation of the original. Simultaneous goal problems occur in mathematical equation solving, in robot tasks, and in real life as well.

An earlier paper (Manna and Waldinger, 1974) proposes a method for dealing with simultaneous goal problems in program synthesis. The present paper elaborates on the description of the method, reports on its implementation, discusses its application to general planning and robot problem solving, and points out some of its shortcomings and some projected improvements.

The general strategy proposed in (Manna and Waldinger, 1974) is: in order to construct a plan to achieve P and Q, construct a plan to achieve P, and then modify that plan to achieve Q as well. In the course of the modification, the relation P is "protected": no modifications that might make P false are permitted. If no satisfactory modification is found, the same strategy is attempted with the roles of P and Q reversed.

The earlier paper considers the construction of programs with branches and recursive loops; here, the discussion is strictly limited to the construction of straight-line programs. The simultaneous goal strategy can be integrated with the branch and loop formation techniques discussed in our earlier paper; however, this integration has not yet been implemented. Furthermore, the straight-line case is rich enough to be interesting in its own right.

The paper is divided into two main parts. Part 1 describes the simultaneous goal strategy in full detail, the program modification technique, and the modelling structure that the strategy requires. The strategy is illustrated by several examples, including the development of programs to interchange the values of two variables and to sort three variables, and the solution of the "anomaly" blocks-world problem from Sussman's (Sussman, 1973) thesis. These examples are not chosen to be impressive; they have been refined to present no difficulties other than the simultaneous goal problem itself.

Part 2 tries to relate this work to some other problem-solving efforts, and provides a critical survey of the way these systems represent a changing world in terms of the framework developed in Part 1. A summary of Part 2 appears in Section 2.9.

PART 1

SIMULTANEOUS GOALS, PROGRAM MODIFICATION, AND THE REPRESENTATION OF ACTIONS

1.1 A description of our approach

1.1.1 Achieving primitive goals

> Below, the boarhound and the boar
> Pursue their pattern as before
> But reconciled among the stars.
>
> T.S. Eliot, *Four Quartets*

Before we are ready to face multiple simultaneous goals, it may be helpful to say a few words about our approach to simple goals. Our system has a number of built-in techniques and knowledge of the kinds of goal to which each technique applies. When faced with a new goal, the system tries to determine if that goal is already true in its model of the world—if the goal is true it is already achieved. Otherwise, the system retrieves those techniques that seem applicable. Each of these techniques is attempted in turn until one of them is successful.

An important clue to the choice of technique is the form of the given goal. For instance, suppose we are working on blocks-world problems, and we are faced with the goal that block A be directly on top of block B. Assume that we have an arm that can move only one block at a time. Then we may build in the following strategy applicable to all goals of form, "Achieve: x is on y": clear the top of x and the top of y, and then put x on y. That x be clear is a new goal, which may already be true in the model, or which may need to be achieved itself (by moving some other block from the top of x to the table, say). "Put x on y" is a step in the plan we are developing. If we can successfully apply this technique, we have developed a plan to put A directly on top of B. However, for a variety of reasons this technique may fail, and then we will have to try another technique.

An example from the program synthesis domain: suppose our goal is to achieve that a variable X have some value b. One approach to goals of this form is to achieve that some other variable v has value b, and then execute the assignment statement X ← v.* Here again, the relation "v has value b" is a subgoal, which may already be true or which may need to be achieved by inserting some other instructions into the plan. The assignment statement X ← v is an operation that this technique itself inserts into the plan. (Note that if we are not careful, this technique will be applicable to its own subgoal, perhaps resulting in an infinite computation.) Of course, there may be other techniques to achieve goals of form "X has value b"; if the original technique fails, the others are applied.

The practice of retrieving techniques according to the form of the goal and then trying them each in turn until one is successful is called "pattern-directed function invocation," after Hewitt (Hewitt, 1972). A problem solver organized around these principles can be aware of only one goal at a time: hence the single-mindedness that Winograd complains of. When given multiple simultaneous goals, we would like to be able to apply the techniques applicable to each goal and somehow combine the results into a single coherent plan that achieves all of them at once.

1.1.2 Goal regression

Change lays not her hand upon truth.

A.C. Swinburne, *Poems: Dedication*

*We use a lower case "v" but an upper case "X" because here, X is the name of a specific variable while v is a symbol that can be instantiated to represent any variable.

Our approach to simultaneous goals depends heavily on having an effective program modification technique. Our program modification technique in turn depends on knowing how our program instructions interact with the relations we use to specify the program's goals.

Suppose P is a relation and F is an action of program instruction; if P is true, and we execute F, then of course we have no guarantee that P will still be true. However, given P, it is always possible to find a relation P′ such that achieving P′ and then executing F guarantees that P will be true afterwards. For example, in a simple blocks world if P is "block C is clear" (meaning C has no blocks on top of it) and F is "Put block A on block B," then P′ is the relation "C is clear or A is on C": for if C is clear before putting A on B, then C will still be clear afterwards, while if A is on C before being put on B, then the action itself will clear the top of C.*

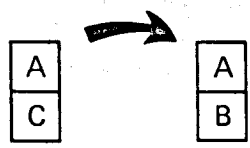


FIGURE 1

We will demand that P′ be the weakest relation that ensures the subsequent truth of P; in other words, if P′ is not true before executing F, P may not be true afterwards. Otherwise, we could always take P′ to be the relation that is always false. We will call P′ the result of passing P back over F, and we will call the operation of passing P back "regression."

Another example: suppose F is an assignment statement "X ← t" where X is a variable and t an expression, and let P be any relation between the values of the variables of our program, written P(X). Then P′ is P(t), the relation obtained by replacing X by t in P(X). For if P(t) is true before executing X ← t, then P(X) will certainly be true afterwards. For instance, if P(X) is "X=A*B," and F is "X ← U*V," then P′=P(U*V) is "U*V=A*B," for if U*V=A*B before executing X ← U*V, then X=A*B afterwards. Furthermore, if U*V=A*B is false before the assignment, then X=A*B will be false afterwards.

Note that if X does not occur in P(X), then P(t) is the same as P(X); the instruction has no effect on the truth of the relation.

Regression will play an important part in our program modification technique and also in the way we construct our models. The use of a static relational description to describe a dynamic program has been variously attributed to (Floyd, 1967), (Naur, 1966), (Turing, 1950), and (Goldstine and von Neumann, 1947), but the observation that it is technically simpler to look at the "weakest preconditions" of a relation (passing it back), as we do, instead of the "strongest

*We assume that the blocks are all the same size, so that only one block can fit immediately on top of another.

postconditions" (passing it forward), appears to be due to (Manna, 1968), (Hoare, 1969), and (King, 1969). The term "weakest precondition" is Dijkstra's (1975); we will not use it because the word "precondition" has a different meaning in the artificial intelligence literature. All these authors apply the idea to proving the correctness of programs; (Manna, 1974) contains a survey of this application. We now go on to show how the idea applies to program modification as well.

1.1.3 Plan modification

It is a bad plan that admits of no modification.

Publilius Syrus, *Sententiae*

In order to achieve a goal of form P and Q, we construct a plan F that achieves P, and then modify F so that it achieves Q while still achieving P. The simplest way to modify F is to add new instructions to the end so as to achieve Q. This method is called a "linear theory plan" by Sussman (Sussman, 1973). However, this linear strategy may be flatly inadequate; for instance, executing the plan F may destroy objects or information necessary to achieve Q. Furthermore, even if Q can be achieved by some composite plan ⟨F:G⟩ (execute F, then execute G), how can we be sure that plan G will not cause P to be made false?

However, we may also modify F by adding new instructions to the beginning or middle, or by changing instructions that are already there. Let us assume that F is a linear sequence of instructions $\langle F_1,...,F_n \rangle$. As we have seen, in order to achieve Q after executing F, it suffices to achieve Q′ immediately before executing F_n, where Q′ is the result of passing Q back over F_n. Similarly, it suffices to achieve Q″ immediately before executing F_{n-1}, where Q″ is the result of passing Q′ back over F_{n-1}.

How can we benefit by passing a goal back over steps in the plan? A goal that is difficult or impossible to achieve after F has been executed may be easier to achieve at some earlier point in the plan. Furthermore, if achieving Q after executing F destroys the truth of P, it is possible that planning to achieve Q′ or Q″ earlier will not disturb P at all; a planner should be free to achieve Q in any of these ways.

How is the planner supposed to know how to pass a relation back over a given plan step? First of all, the information can be given explicitly, as one of a set of rules. These "regression rules," which can themselves be expressed as programs, are regarded as part of the definition of the plan step. Alternatively, if a relation is defined in terms of other relations, it may be possible to pass back those defining relations. Furthermore, if the plan step is defined in terms of simpler component plan steps, then knowing how to pass relations back over the components allows one to pass the relation back over the original plan step. Finally, if no information at all exists as to how to pass a relation back over a plan step, it is assumed that the plan step has absolutely no effect on the relation. This assumption makes it unnecessary to state a large number of rules, each saying that a certain action has no effect at all on a certain relation. Thus we avoid the

so-called "frame problem" (cf. [McCarthy and Hayes, 1969]).

In modifying a program it is necessary to ensure that it still achieves the purpose for which it was originally intended. This task is performed by the protection mechanism we will now describe.

1.1.4 Protection

> Protection is not a principle, but an expedient.
>
> Disraeli, Speech

Our strategy for achieving two goals P and Q simultaneously requires that after developing a plan F that achieves P we modify F so that it achieves Q while still achieving P. This strategy requires that in the course of modifying F the system should remember that F was originally intended to achieve P and check that it still does. It does this by means of a device called the protection point: we attach P to the end of F as a comment. This comment has imperative force: no modifications are permitted in F that do not preserve the truth of P at the end of the modified plan. We will say that we are protecting P at the end of F. Any action that destroys the truth of P will be said to violate P. Relations may be protected at any point in a plan; if a relation is protected at a certain point, that relation must be true when control passes through that point.*

Protection has purposes other than ensuring that simultaneous goals do not interfere with each other: for instance, if an action requires that a certain condition be true before it can be applied, we must protect that condition at the point before the action is taken to see that no modification in the plan can violate it.

In order to ensure that a modification cannot violate any of the protected relations, we check each of these relations to see that it is still true after the proposed modification has been made: otherwise, the modification must be retracted.

In the next section we will examine a very simple example involving two simultaneous goals in order to demonstrate the techniques we have described.

1.1.5 A very simple example

Suppose we have three blocks, A, B, and C, sitting on a table.

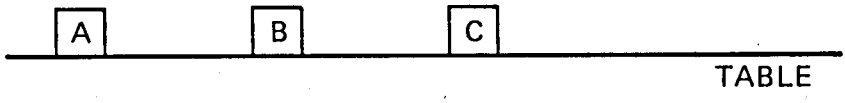


FIGURE 2

*(Sussman, 1973) was the first to use protection in program synthesis, and to apply it to the simultaneous goal problem.

Our goal is to make a tower of the three blocks, with A on top and C on the bottom.

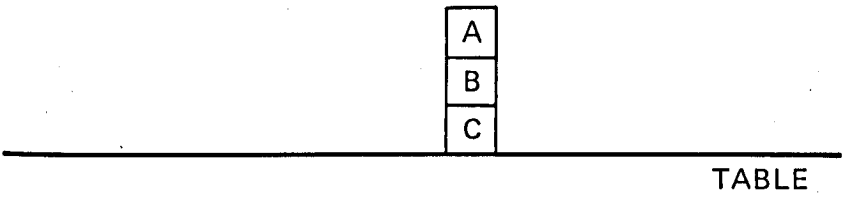


FIGURE 3

We express this goal as a conjunction of two goals. "A is on B" and "B is on C." (We'll forget about saying that C is on the table.) Of course, if we approach these goals in the reverse order we have no problem: we simply put B on top of C and then put A on top of B; no destructive interactions arise. However, if we approach them in the given order we run into a blind alley.

We first attempt to achieve that A is on top of B. In order to do this, we see if A and B are clear (they are), and then we plan to put A on top of B. We have thus planned to achieve our first goal. Because we will now work on another goal to be achieved simultaneously we protect the relation that A is on top of B. We will adopt a notation for representing plans under development in which the left-most column will represent the steps of the plan, the second column will represent the anticipated model or state of the world between the respective plan steps, and the third column will represent any goals that we have yet to achieve, and relations that have already been achieved but must be protected at that point. In this notation our plan so far is as follows:

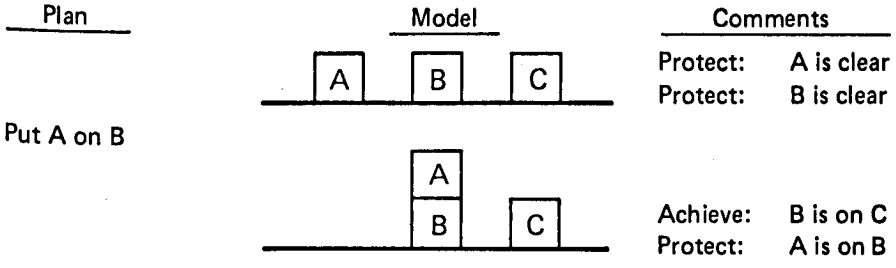


FIGURE 4

In order to put A on top of B we must be sure that A and B are both clear: therefore we have protected these two relations at the point before the action is applied. (Of course, the action itself violates one of the conditions afterwards: we merely want to ensure that the conditions will be true immediately before the action is applied, regardless of what modifications are made to the plan.) We put the goal "Achieve: B is on C" after the plan step and not before because we

are initially attempting to achieve the goal by adding steps to the end of the plan and not the beginning.

Now, since our arm can lift only one block at a time, we will be forced to put A back on the table again in order to get B on top of C. This will violate our protected relation (A is on B) so we cannot hope to achieve our second goal by adding instructions to the end of the plan. But we can still try to pass the goal back over the plan. The goal "B is on C" passed back over the plan "Put A on B" is simply "B is on C" itself, because putting A on B will not alter whether or not B is on C. The plan state so far is as follows:

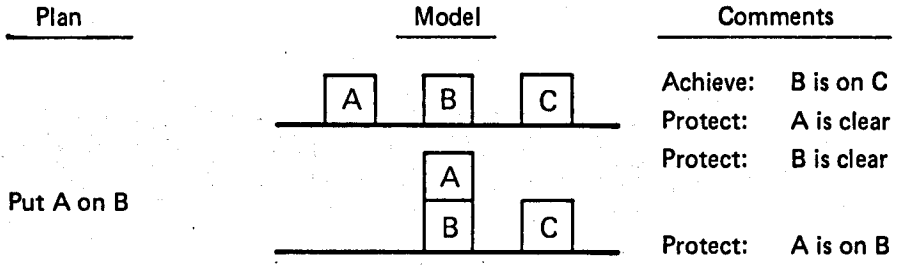


FIGURE 5

The goal "Achieve: B is on C" now occurs before the plan step.

Our goal "B in on C" can now be achieved by simply putting B on C; the appropriate plan step will be added to the beginning of the plan instead of to the end. The resulting plan state is illustrated in Figure 6.

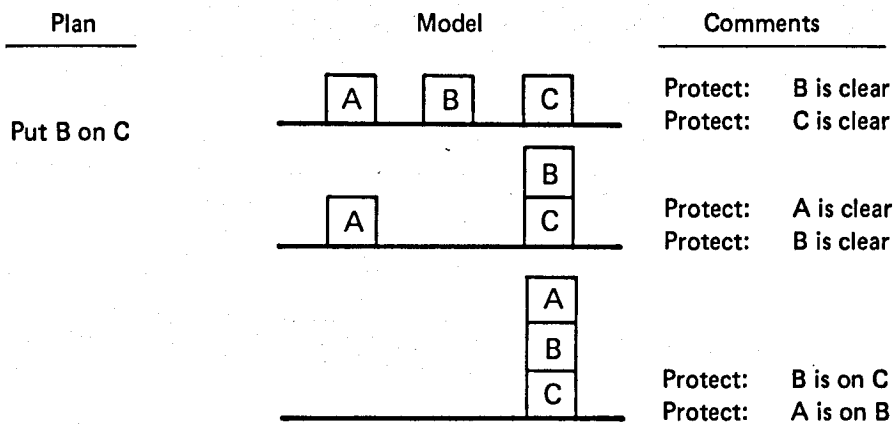


FIGURE 6

Note that the new plan step did not interfere with any of the protected relations: otherwise we would have had to retract the step and find some other solution. As it is, the two-step plan "Put B on C; Put A on B" achieves the desired goal. The method of passing goals back over plan steps has enabled us to

avoid backing up and reversing the order in which the goals are approached. This technique will not always prevent such goal reordering; however, we will see that it will allow us to solve some problems that cannot readily be solved, regardless of the order in which the goals are attempted.

The reader may note that the model following the plan step "Put A on B" changes between Figure 5 and Figure 6, because of the insertion of the earlier plan step "Put B on C." It we maintained a model corresponding to each plan step, we would be faced with the task of updating the entire sequence of models following every insertion to reflect the action of the new plan step. This can be an arduous chore if the model is at all large. Instead we maintain only a scanty "skeleton" model that is not affected by an alteration, and generate or "flesh out" other portions of the model as needed, using the same regression method that we introduced earlier as a program modification technique.

1.1.6 Skeleton models

Following each step in the developing plan we have a model, which for our purposes may be regarded as a list of relations that are certain to be true following the execution of that plan step. For instance, following the step "Put A on B" we include in the model the relation "A is on B" and perhaps the relation "A is clear," meaning that no block is on top of A. However, we do not usually include any information about the location of B, for example, because, unless protected, the location of B can be changed by inserting new steps earlier in the plan.

Similarly, after an assignment statement $X \leftarrow t$ we do no generally include the fact that X has value 2 even if we believe that t has value 2 before the statement is executed, because subsequent modifications to the beginning of the program could change the value of t, unless that value is protected. In fact, the model following an assignment statement may be absolutely empty.

In addition to the models that follow each statement in the plan, we have an initial model that describes that situation at the beginning (as given by the problem statement), and we have a global model of the "eternal verities," relations such as x=x, that are unchanged by any action or the passage of time. Information in the global model is implicitly present in all the other models.

The models that follow each action in the plan are incomplete: much knowledge about the situation is not included explicitly. How are we to compensate for this deficiency?

Suppose that we are given a plan $F_1,...,F_n$, and we need to know whether some relation Q is true after execution of step F_i. We first see if Q is explicitly in the model following F_i; in other words, we see if Q is an immediate consequence of the execution of F_1. If not, we simply pass Q back over the plan step F_i, yielding a perhaps altered relation Q'. We then check if Q' is in the preceding model. The justification for this measure is clear: Q' has been defined as the relation that must be true before the execution of F_i in order that Q will be true afterwards.

If we fail to determine if Q' is true, we pass Q' back over F_{i-1} and repeat the

process until we have passed Q all the way back to the intitial model. If we are still unable to determine whether Q is true we must give up. Even if we determine that Q is true, we must generally resist the temptation to add it to the model that follows F_i: unless Q is protected, later plan alterations could make Q false, and then the model would be inaccurate.

An example: suppose we are given a model in which block A is on C, but blocks A and B both have a clear top.

FIGURE 7

We somehow develop the plan step "Put A on B," and we are led to inquire if C is clear. We cannot determine this from the model that follows "Put A on B," because that model only contains the relations "A is on B" and "A is clear." However, we can pass that relation back over the plan step using a regression rule (as described in Section 1.1.2), leading us to ask if "C is clear or A is on C." Since we know "A is on C" initially, we can conclude "C is clear" in the model following the plan step.

The skeleton model is a technique in which the partial plan that has been constructed is regarded as a central part of the model. Important relationships and the plan itself are in the model explicitly; other relationships may be inferred using the regression rules.

It is traditional in problem solving to distinguish between rules that work backwards from the goal and rules that work forwards from the present state of the world. In Hewitt's (Hewitt, 1972) terminology, these rules are called "consequent theorems" and "antecedent theorems" respectively. Regression rules are a special kind of consequent theorem that can refer explicitly to steps in the plan as well as relations in the model. (Kowalski, 1974) and (Warren, 1974) also discuss the application of regression rules as a modelling technique.

The use of skeleton models means that if a relation P is protected at the end of a plan, no modification can be made at any point in the plan that will not leave P true at the end, because, in checking the truth of P after the modification has been made, we will percolate P back up through the plan, and the unfortunate interaction between P and the new plan step will be discovered.

For instance, suppose a plan step $X \leftarrow Y$ achieves a protected relation P(X), and a new instruction $Y \leftarrow Z$ is inserted at the beginning of the plan, where P(Z) is false. We will try to check that the protected relation P(X) is still true at the end of the modified program. Using regression, we will therefore check if P(Y) is true in the middle of the program, and thus that P(Z) is true at the beginning. Since P(Z) is false, we will detect a protection violation and reject the proposed modification.

This mechanism means that it is necessary to protect a relation only at the point at which we need it to be true. In the previous example, we must protect P(X) after the assignment statement X ← Y, but we need not protect P(Y) before the statement; the latter protection is implicit in the former.

A description of how skeleton models can be implemented using the "context" mechanism of the new artificial intelligence programming languages occurs in Section 2.7.

We have concluded the general description of our approach to simultaneous goals. The balance of Part 1 concerns how this technique has been applied to specific subject domains in order to solve the sample problems.

1.2 Interchanging the values of two variables

1.2.1 Relations that refer to variables

> So first, your memory I'll jog,
> And say: A CAT IS NOT A DOG.
>
> T.S. Eliot, *Old Possum's Book of Practical Cats*

In the next section we will show the synthesis of a more complex program whose specification is represented as a set of simultaneous goals. The subject domain of this program will be variables and their values. However, we must first examine a certain kind of relation more closely: the relation that refers directly to the variable itself, as opposed to its value. For instance, the relation "variable X has value a," written "X:a," refers both to the variable X and its value a. The relations "variable X is identical to variable Y," written "X≈Y," and its negation "variable X is distinct from variable Y," written "X≉Y" refer to variables X and Y, but do not refer at all to their values. X≉Y means "X and Y are not identical," and is true regardless of whether X and Y have the same value. Relations such as ≈, which do not refer to values at all, are not affected by assignment statements or any program instructions we are going to consider. Relations such as ":" are more complicated. For instance, the relation X:a passed back over the assignment statement X ← Y yields Y:a, where X and Y are both variables. (A more general rule covers the case in which an arbitrary term plays the role of the variable Y, but we will have no need to consider this case in the following examples.) A more complex situation arises if the variable in the relation is existentially quantified. Such a situation arises if the relation is a goal to find a variable with a certain value. For instance, how do we pass back a goal such as "Find a variable v such that v:a" over the instruction X ← Y? If there is a variable v such that v:a before the assignment statement is executed, and if that variable is distinct from X, then certainly v:a after the execution of X ← Y. Furthermore, if Y:a before the execution, then v can be identical to X as well. Therefore, passing the goal "Find a variable v such that v:a" back over the assignment statement X ← Y yields

"Find a variable v such that
$v \not\approx X$ and v:a
or $v \approx X$ and Y:a."

We will assume the system knows verities such as $x \approx x$, $X \not\approx Y$, or $X \not\approx Z$. In the example of the next section we will use one additional fact about the relation $\not\approx$: the fact that we can always invent a new variable. In particular, we will assume we can find a variable v such that $v \not\approx X$ by taking v to be the value of a program GENSYM that invents a new symbol every time it is called.

There is, of course, much more to be said about these peculiar relations that refer to variables themselves. They do not follow the usual Floyd-Naur-Manna-King-Hoare rule for the assignment statement. However, the discussion in this section will be enough to carry us through our next example.

1.2.2 The solution to the two variable problem*

> But above and beyond there's still one name left over,
> And that is the name that you never will guess
> The name that no human research can discover–
> But THE CAT HIMSELF KNOWS, and will never confess.
>
> T.S. Eliot, *Old Possum's Book of Practical Cats*

The problem of exchanging the values of two variables is a common beginner's programming example. It is difficult because it requires the use of a "temporary" variable for storage. Part of the interest of this synthesis involves the system itself originating the idea of using a generated variable for temporary storage.

We are given two variables X and Y, whose initial values are a and b; in other words, X:a and Y:b. Our goal is to produce a program that achieves X:b and Y:a simultaneously.

Recall that our strategy when faced with a goal P and Q is to try to form a plan to achieve P, and then to modify that plan to achieve Q as well. Thus our first step is to form a plan to achieve X:b.

For a goal of form X:b we have a technique (Section 1.1.1) that tells us to find a variable v such that v:b and then execute the assignment statement $X \leftarrow v$. We have such a v, namely Y. Therefore, we develop a plan, $X \leftarrow Y$, that achieves X:b. We must now modify this plan to achieve Y:a while protecting the relation X:b that the plan was developed to achieve. In our tabular notation:

---

*Another way of approaching this problem is discussed in (Green *et al.*, 1974). Green's system has the concept of temporary variable built in. He uses a convention of inserting a comment whenever information is destroyed, so that a patch can be inserted later in case the destroyed information turns out to be important.

Plan	Model	Comments
	X:a Y:b	
X ← Y		
		Achieve: Y:a
	X:b Y:b	Protect: X:b

FIGURE 8

(In our table we record the full model at each stage even though the implementation does not store this model explicitly.

In trying to achieve Y:a we attempt to find a variable v such that v:a. Once we have executed X ← Y, no such variable exists. However, we pass the goal "Find v such that v:a" back over the plan step X ← Y, yielding

Find v such that
v≠X and v:a
or v≈X and Y:a,

as explained in the preceding section. We now attempt to achieve this goal at the beginning of the plan. In tabular form

Plan	Model	Comments
	X:a Y:b	Achieve: Find v such that
		v≠X and v:a
		or v≈X and Y:a
X ← Y		
	X:b Y:b	Protect: X:b

FIGURE 9

Once the outstanding goal is achieved, we will add an assignment statement Y ← v to the end of the program, where v is the variable that achieves the goal.

If we work on the goals in the given order, we try to find a v such that v≠X. Here we know that GENSYM will give us a new variable name, say G_1, guaranteed to be distinct from X. Our problem is now to achieve the first conjunct, namely G_1:a. But this can easily be achieved by inserting the assignment statement G_1 ← X at the beginning of the plan, since X:a initially. Inserting this instruction does not disturb our protected relation.

We have been trying to find a v satisfying the disjunction

v≠X and v:a
or v≈X and Y:a

We have satisfied the first disjunct, and therefore we can ignore the second. (We

will consider later what happens if we reverse the order in which we approach some of the subgoals.)

We have thus managed to find a v such that v:a at the end of the program, namely $v \approx G_1$. Since our ultimate purpose in finding such a v was to achieve Y:a, we append to our program the assignment statement $Y \leftarrow G_1$. This addition violates no protected relations, and achieves the last of the extant goals. The final program is thus

Plan	Model	Comments
	X:a Y:b	
$G_1 \leftarrow X$		
	X:a Y:b G_1:a	
$X \leftarrow Y$		
	X:b Y:b G_1:a	
$Y \leftarrow G_1$		
		Protect: Y:a
	X:b Y:a G_1:a	Protect: X:b

FIGURE 10

The program has "invented" the concept of "temporary variable" by combining two pieces of already existing knowledge: the fact that GENSYM produces a variable distinct from any given variable, and the rule for passing a goal "Find a v such that v:a" back over an assignment statement. Of course, we could have built in the temporary variable concept itself, and then the solution would have been found more easily. But in this case the invention process is of more interest than the task itself.

Notice that at no point in the construction did we violate a protected relation. This is because of the fortunate order in which we have approached our subgoals. For example, if we had chosen to work on the disjunct

$$v \approx X \text{ and } Y{:}a$$

instead of

$$v \not\approx X \text{ and } v{:}a,$$

we would have inserted the assignment statement $Y \leftarrow X$ at the beginning of the program in order to achieve Y:a, and we would have proposed the program

$$\begin{array}{l} Y \leftarrow X \\ X \leftarrow Y \\ Y \leftarrow X \end{array}$$

which violates the protected relation X:b. Other alternative choices in this

synthesis are either successful or terminated with equal dispatch.

1.3 Sorting three variables*

1.3.1 Sorting two variables

In our next example we will see how to construct a program to sort the values of three variables. This program will use as a primitive the instruction sort2, which sorts the values of two variables. Before we can proceed with the example, therefore, we must consider how to pass a relation back over the instruction sort2.

Executing sort2(X Y) will leave X and Y unchanged if X is less than or equal to Y ($X \leqslant Y$), but will interchange the values of X and Y otherwise. Let P(X Y) be any relation between the values of X and Y. We must construct a relation P'(X Y) such that if P'(X Y) is true before sorting X and Y, P(X Y) will be true afterwards. Clearly, if $X \leqslant Y$, it suffices to know that P(X Y) itself is true before sorting, because the sorting operation will not change the values. On the other hand, if $Y<X$ it suffices to know P(Y X), the expression derived from P(X Y) by exchanging X and Y, because the values of X and Y will be interchanged by the sorting. Therefore, the relation P'(X Y) is the conjunction

$$\begin{array}{ll} & \text{if } X \leqslant Y \text{ then } P(X\ Y) \\ \text{and} & \text{if } Y<X \text{ then } P(Y\ X) \end{array}$$

A similar argument shows that the above P' is as weak as possible. The same line of reasoning applies even if X or Y does not actually occur in P. For instance, if X does not occur, P(Y X) is simply P(X Y) with Y replaced by X.

Given the appropriate definition of sort2, it is straightforward to derive the above relation mechanically (e.g., see [Manna, 1974]). However, that would require the system to know about conditional expressions, and we do not wish to discuss those statements here. For our purposes, it suffices to assume that the system knows explicitly how to pass a relation back over a sort2 instruction.

1.3.2 Achieving an implication

We have excluded the use of conditionals in the programs we construct. However, we cannot afford to exclude the goals of form "if P then Q" from the specifications for the program being constructed. For instance, such specifications can be introduced by passing any relation back over a sort2 instruction.

*This problem is also discussed in (Green, *et al.*, 1974). Green allows the use of program branches and the program he derives has the form of a nested conditional statement. Green's use of the case analysis avoids any protection violations in his solution: the interaction between the subgoals plays a much lesser role in Green's formulation of the problem. Some other work in the synthesis of sort programs (see [Green and Barstow, 1975], [Darlington, 1975]) does not consider "in-place" sorts at all; goal interactions are still important, but protection issues of the type we are considering do not arise. However, Darlington's concept of "pushing in" a function is the analogue of regression for programs in which nested functional terms play the role of sequential program instructions.

The form of these specifications suggests that the forbidden conditional expression be used in achieving them. Therefore, for purposes of this example we will introduce a particularly simple-minded strategy: to achieve a goal of form "if P then Q," first test if P is known to be false: if so, the goal is already achieved. Otherwise, assume P is true and attempt to achieve Q.

The strategy is simple-minded because it does not allow the program being constructed to itself test whether P is true; a more sophisticated strategy would produce a conditional expression, and the resulting program would be more efficient. However, the simple strategy will carry us through our next example.

1.3.3 The solution to the three-sort problem

Given three variables, X, Y, and Z, we want to rearrange their values so that $X \leqslant Y$ and $Y \leqslant Z$. Either of these goals can be achieved independently, by executing sort2(X Y) or sort2(Y Z) respectively. However, the simple linear strategy of concatenating these two instructions does not work; the program

sort2(X Y)
sort2(Y Z)

will not sort X, Y, and Z if Z is initially the smallest of the three. On the other hand, the simultaneous goal strategy we have introduced does work in a straightforward way.

In order to apply our strategy, we first achieve one of our goals, say $X \leqslant Y$, using the primitive instruction sort2(X Y). We then try to modify our program to achieve $Y \leqslant Z$ as well. In modifying the program we protect the relation $X \leqslant Y$. In tabular form, the situation is as follows:

Plan	Model	Comments
sort2(X Y)		
		Achieve: $Y \leqslant Z$
	$X \leqslant Y$	Protect: $X \leqslant Y$

FIGURE 11

As we have pointed out, simply appending a plan step sort2(Y Z) will violate the protected relation $X \leqslant Y$. Therefore we pass the goal back to see if we can achieve it at an earlier stage. The regressed relation, as explained in the previous section, is

if $X \leqslant Y$ then $Y \leqslant Z$
and if $Y < X$ then $X \leqslant Z$.

(This relation effectively states that Z is the largest of the three numbers.) Our situation therefore is as follows:

Plan	Model	Comments
		Achieve: if $X \leqslant Y$ then $Y \leqslant Z$ and if $Y < X$ then $X \leqslant Z$
sort2(X Y)		
	$X \leqslant Y$	Protect: $X \leqslant Y$

FIGURE 12

We must now try to achieve the remaining goal. This goal is itself a conjunction and is handled by the simultaneous goal strategy. The first conjunct, "if $X \leqslant Y$ then $Y \leqslant Z$," is an implication. Therefore we first test to see if $X \leqslant Y$ might be known to be false, in which case the implication would be true. However, nothing is known about whether $X \leqslant Y$, so we assume it to be true and resign ourselves to achieving the consequent $Y \leqslant Z$: this can easily be done using the primitive instruction sort2(Y Z). Inserting this instruction at the beginning of the plan does not interfere with the protected relation $X \leqslant Y$: the protection point is immediately preceded by the instruction sort2(X Y). Our situation is therefore as follows:

Plan	Model	Comments
sort2(Y Z)		
		Achieve: if $Y < X$ then $X \leqslant Z$
	$Y \leqslant Z$	Protect: if $X \leqslant Y$ then $Y \leqslant Z$
sort2(X Y)		
	$X \leqslant Y$	Protect: $X \leqslant Y$

FIGURE 13

(Notice that we do not reproduce the complete model for this example, but only include the skeleton model.)

We have achieved the goal "if $X \leqslant Y$ then $Y \leqslant Z$," which is one of two simultaneous goals. We therefore protect the relation we have just achieved and attempt to modify the program to achieve the remaining goal, "if $Y < X$ then $X \leqslant Z$." Again, we cannot disprove $Y < X$ and therefore we attempt to achieve the consequent, $X \leqslant Z$. This goal can be achieved immediately by executing sort2(X Z), but we must check that none of the protected relations is disturbed. Our situation is

Plan	Model	Comments
sort2(Y Z)		
	$Y \leqslant Z$	
sort2(X Z)		
	$X \leqslant Z$	Protect: if $X \leqslant Y$ then $Y \leqslant Z$
sort2(X Y)		
	$X \leqslant Y$	Protect: $X \leqslant Y$

FIGURE 14

The second protected relation $X \leqslant Y$ is still preserved: the first presents us with a bit more difficulty, but is in fact true: a human might notice that Z is the largest of the three numbers at this point. Perhaps it is worth explaining how the system verifies this protected relation, thereby illustrating the use of the skeleton model.

After executing the second instruction sort2(X Z), the only information in skeleton model is that $X \leqslant Z$. This is not enough to establish that the protected relation is undisturbed. The system therefore passes the relation back to an earlier model and tries to prove it there. The regressed relation is

$$\begin{aligned} & \text{if } X \leqslant Z \text{ then (if } X \leqslant Y \text{ then } Y \leqslant Z) \\ \text{and} \quad & \text{if } Z < X \text{ then (if } Z \leqslant Y \text{ then } Y \leqslant X). \end{aligned}$$

The earlier model tells us that $Y \leqslant Z$ [because we have just executed sort2(Y Z)]. The first conjunct is thus easy to prove: the conclusion $Y \leqslant Z$ is known explicitly by the model. The second conjunct follows from transitivity: since we know $Y \leqslant Z$ from the model and $Z < X$ from the hypothesis we can conclude that $Y \leqslant X$. (This sort of reasoning is performed by a mechansim described in [Waldinger and Levitt, 1974]). The program in Figure 14 is therefore correct as it stands (although additional relationships should be protected if the plan is to undergo further modification).

It is pleasing that this last bit of deduction was not noticed by Manna and Waldinger in preparing the 1974 paper, but was an original discovery of the program, which was implemented afterwards. Manna and Waldinger assumed the protected relation would be violated and went through a somewhat longer process to arrive at an equivalent program. This is one of those not-so-rare cases in which a program debugs its programmer.

In order to show how these ideas apply to robot-type problems we discuss one further example, Sussman's "anomaly," in the next section.

1.4 The Sussman "anomaly"

We include this problem because it has received a good deal of attention in the robot planning literature (e.g., [Sussman, 1973; Warren, 1974; Tate, 1974; Hewitt, 1975; Sacerdoti, 1975]). However, for reasons that we will explore in Part 2, the solution does not exercise the capabilities of the system as fully as the previous two examples. We are given three blocks in the following configuration:

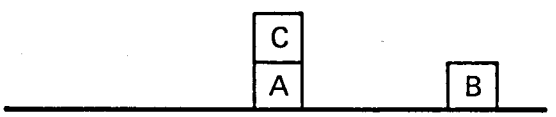


FIGURE 15

We are asked to rearrange them into this configuration:*

FIGURE 16

The goal is thus a simple conjunction "A is on B and B is on C." (We will forget about the table.)

The anomaly is one of the simplest blocks-world problems for which the linear strategy does not work regardless of the order in which we approach the subgoals: if we clear A and put A on B we cannot put B on C without removing A:

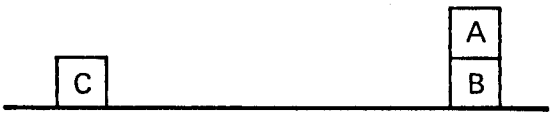


FIGURE 17

(Remember the arm can lift only one block at a time.)

On the other hand, if we put B on C first, we have buried A and cannot put it on top of B without disturbing the other blocks:

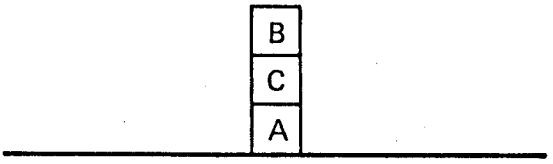


FIGURE 18

Our technique can solve this problem regardless of the order in which it attacks the goals. We will consider just one of these orderings: Assume we attempt to achieve "A is on B." The system will generate subgoals to clear A and B. B is already clear, and A will be cleared by putting C on the table. Then A will be put on B. This much can be done by the elementary strategy for achieving the "on" relationship (Section 1.1.1). Our situation is as follows:

*This problem was proposed by Allan Brown. Perhaps many children thought of it earlier but did not recognize that it was hard.

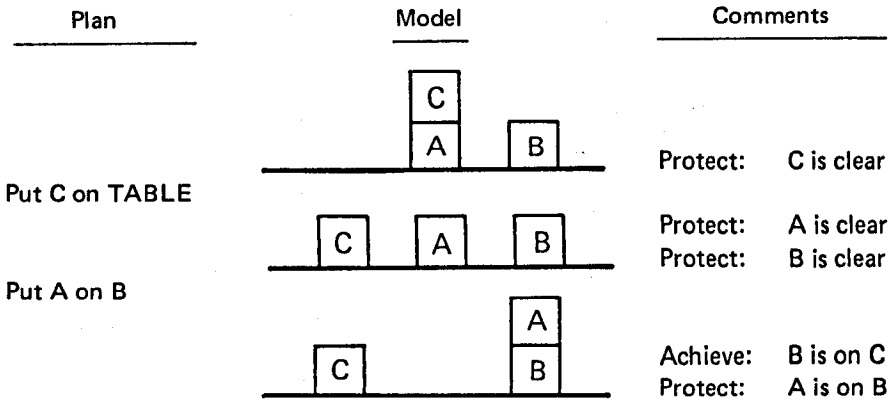


FIGURE 19

We protect "A is on B" because we want to modify the plan to achieve "B is on C" while still achieving "A is on B." We protect "A is clear" and "B is clear" earlier in order to make sure that the operation "Put A on B" will still be legal after the modifications are made.

Now, we have seen that we cannot achieve "B is on C" by adding new steps to the end of the plan without disturbing the protected relation "A is on B." Therefore we again pass the goal back to an earlier stage in the plan, hoping to achieve it before the protected relationship is established.

Passing "B is on C" back over the plan step "Put A on B" yields "B is on C" itself: whether B is on C or not is unaffected by putting A on B. The situation is thus:

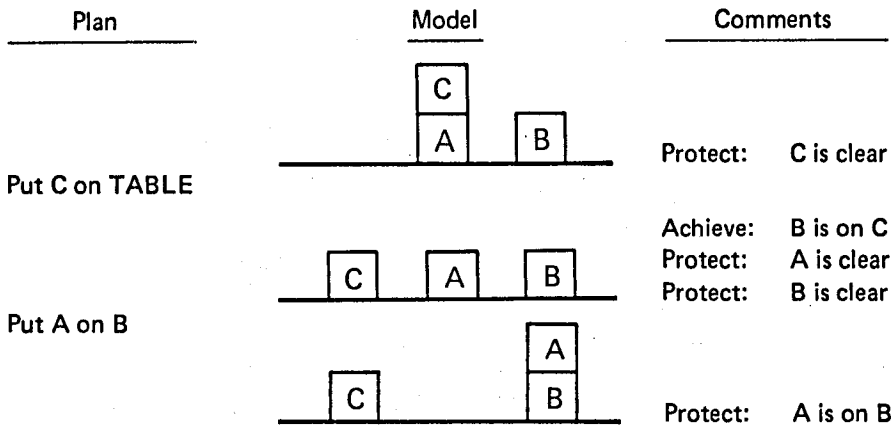


FIGURE 20

The goal "B is on C" can be easily achieved at the earlier stage: B and C are both clear, so we can simply put B on C. Furthermore this operation does not

violate any of the protected relations. Since all goals have been achieved, our final plan is as follows:

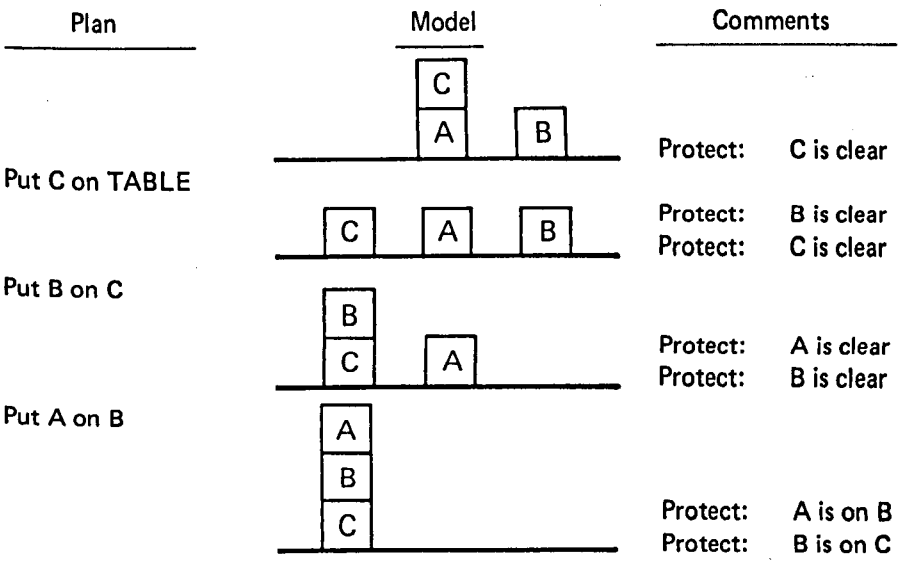


FIGURE 21

The solution is similar if the order in which the goals are attempted is reversed.

This completes the last of our examples. In the next section we discuss some of the limitations of this approach, and consider how they might be transcended.

1.5 Limitations and next steps

> Odin. . .of all powers mightiest far art thou
> Lord over men of Earth, and Gods in heaven,
> Yet even from thee thyself hath been withheld.
> One thing: to undo what thou thyself hast ruled.
>
> Matthew Arnold, *Balder Dead*

The policy maintained by our implementation is to allow no protection violations at all: if a proposed modification causes a violation, that modification is rejected. This policy is a bit rigid and can sometimes inhibit the search for a solution.

For instance, consider the blocks problem in which initially the blocks are as follows:

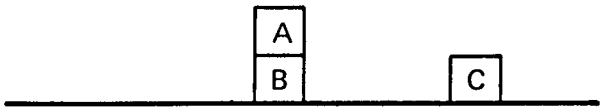


FIGURE 22

and in which the goal is to construct the following stack:

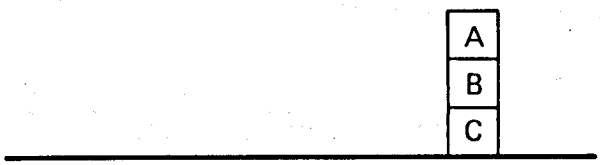


FIGURE 23

The goal may be considered to be the conjunction of two goals, "A is on B" and "B is on C." If these goals are approached in the reverse order, the system has no problem: it clears B by putting A on the table, puts B on C and then puts A on B. However, if the system approaches the goals in the given order, it will attempt to achieve "A is on B" first. This relation is already true, so the system protects it while trying to achieve the goal "B is on C." Here the system is baffled: it cannot put B on C without clearing B, thereby violating the protected relation. Passing the goal backwards into the plan is of no use: there are no plan steps to back it over. Clearly we would like to relax the restriction against protection violation until B is safely on C, and then reachieve the relation "A is on B," but our policy does not permit such a maneuver. The system is forced to reorder the goals in order to find a solution.

The restriction against violating protected relations also lengthens the search in generating the program to sort three variables. If these violations were permitted, a correct program

sort2(X Y)
sort2(Y Z)
sort2(X Y)

could be constructed without the use of regression at all. Why not permit violations, under the condition that a "contract" is maintained to reachieve protected relations that have been violated?

Indeed, such a strategy is quite natural, but we have two objections to it. First, suppose in the course of reachieving one protected relation we violate another. Are we to reachieve that relation later as well, and so on, perhaps ad infinitum? For example, in searching for a plan to reverse the contents of two variables it is possible to generate the infinite sequence of plans

X←Y,
Y←X
X←Y,
X←Y
Y←X
X←Y,
Y←X
X←Y
Y←X
X←Y,...

Each plan corrects a protection violation perpetrated by the previous plan—but commits an equally heinous violation itself. (This objection is a bit naive: one could invent safeguards against such aberrations, as has been done by Sussman (Sussman, 1973) and Green et al. (Green et al., 1974).

The second objection: allowing temporary protection violations can result in inefficient plans. For example, we could generate the following plan for solving the Sussman anomaly:

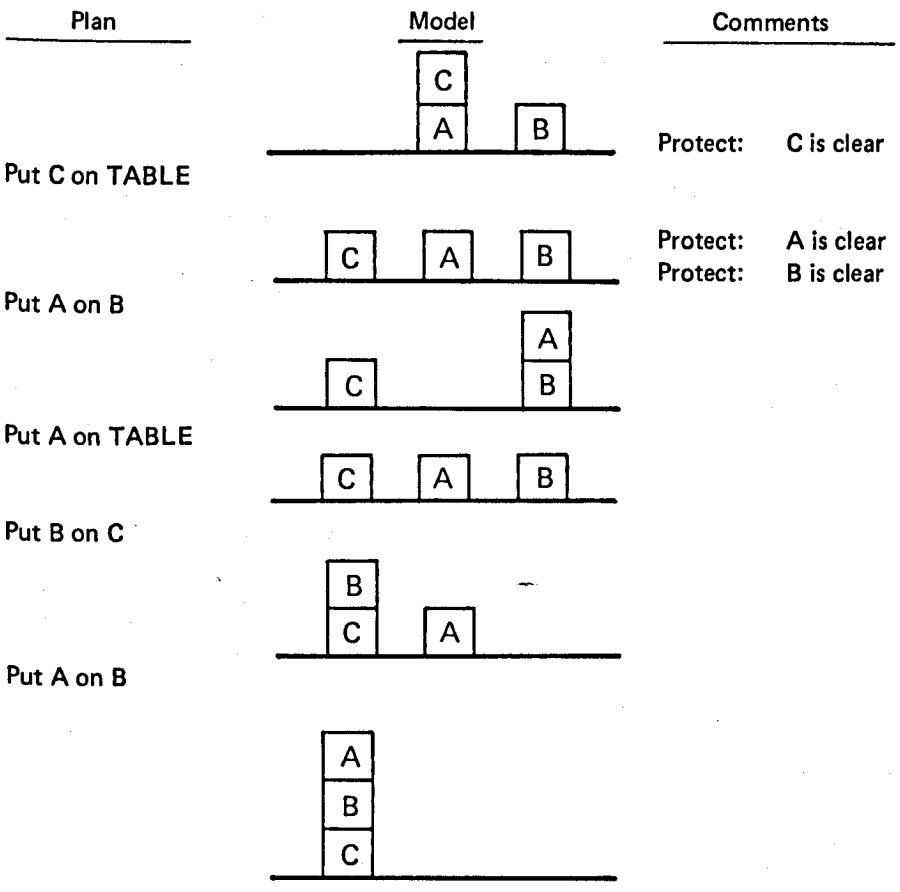


FIGURE 24

This plan is correct but inefficient: We have put A on B only to put A back on the table again because a protection violation was temporarily admitted. In a similar way, Sussman's HACKER produces an equally inefficient plan, approaching the goals in the opposite order. Of course, the plans could later be optimized, but allowing protection violations seems to encourage inefficiency in the plan produced.

Nevertheless, we feel that permitting temporary protection violations in a controlled way is a natural strategy that may be admitted in future versions of the program.

A more serious limitation of our implementation is that the only way it can modify plans is by adding new instructions, never by changing instructions that are already there. For example, suppose we have the initial configuration

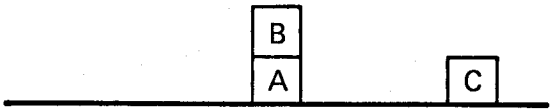


FIGURE 25

and our goal is to construct the stack

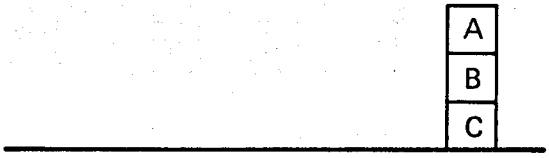


FIGURE 26

Assuming we approach the goal "A is on B" first, we are quite likely to put B on the table and then put A on B. In modifying the plan to achieve "B is on C," it would be clever to plan to put B on C instead of the table, but this sort of modification is beyong the system's capabilities. The "formal object" approach of Sussman (Sussman, 1973) would handle this properly: there, the decision about where to put B (in clearing A) would be deferred until we attempted the second goal "B is on C." However, other sorts of modifications require achieving the same goals in entirely different ways in order to accomodate the demands of the additional specification. Certain protected relations might never be achieved at all in the modified program if the higher level goal that constituted the "purpose" of the protected subgoal could be achieved in some other way. To effect such modifications will require that in the course of modifying a program we retain some of the goal-subgoal hierarchy that caused the original program to be constructed. Such modifications are in the spirit of our approach, but beyond the capabilities of our simple implementation.

The plans we have constructed in our paper are "straight-line" programs; they involve no loops or branches. The system as it exists contains a subsystem for constructing programs with branches and recursive loops (cf. [Manna and Waldinger, 1974]); however, these programs are free of side effects. Since the mechanisms for loop branch construction have not been integrated with the system that constructs structure-altering programs of the sort we have discussed in this paper. Nevertheless, these mechanisms are entirely consistent, and we

intend to unite them. Our hand simulations indicate that the system will then be able to construct a variety of array-sorting routines.

The use of goal regression for these more complex programs has been studied by many as a way of proving programs correct. Passing relations back into branches is straightforward (Floyd, 1967, Hoare 1969); passing a relation back into a loop, on the other hand, may require ingenuity to generalize the relation. This problem is discussed by (Katz and Manna, 1973; Wegbreit, 1974; Boyer and Moore, 1973; Moore, 1975) and others, but it is by no means "solved."

All the loops constructed by our synthesizer will initially be recursive: we intend to introduce iteration only during a subsequent optimization phase, following (Darlington and Burstall, 1973).

The way we have implemented skeleton modelling may be remarkably inefficient, particularly if the plan being constructed is to have many steps. It may take a long time to pass a relation back so far, and the transformed relation may grow alarmingly. There are many ways one might consider to make skeleton modelling more efficient. We prefer not to speculate on which of these ways will actually help until we have tried to implement some of them.

We regard program modification as a valuable synthesis technique apart from its role in achieving goals simultaneously. Often we can construct a program by modifying another program that achieves a goal that is somehow similar or analogous. For instance, in (Manna and Waldinger, 1974) we show how a unification algorithm could be constructed by modifying a pattern matcher. Another sort of program modification is optimization: here we try to modify the program to achieve the same goal more efficiently. It is our hope that systems with the ability to modify their own programs will be able to adapt to new situations without needing to be "general." Before that can happen, however, program modification techniques must be developed beyond what has been done here.

This concludes our discussion of the simultaneous goal strategy. In the next part of this paper we discuss how some other problem solvers have approached some of the same problems.

PART 2

THE REPRESENTATION OF ACTIONS AND SITUATIONS IN CONTEMPORARY PROBLEM SOLVING

> Time present and time past
> Are both perhaps present in time future,
> And time future contained in time past.
> If all time is eternally present
> All time is unredeemable.
>
> T.S. Eliot, *Four Quartets*

In the rest of this paper we will examine a number of problem-solving systems, asking the same question of each system: how are actions and their effects on the world represented? Thus we will not emphasize simultaneous goals

in this section, and in discussing a system we will often ignore the very facets that make it unusual. Many of these systems approach problems of far greater complexity than those we have addressed in Part 2, problems involved in manipulating many more objects, and more complex structures. When we compare our approach to theirs, please bear in mind that our implementation has not been extended to handle the problems that our hand simulation dispatches with such ease.

2.1 The classical problem solvers

In the General Problem Solver (GPS) (see [Newell, Shaw, and Simon, 1960]), the various states of the world were completely independent. For each state, GPS had to construct a new model: no information from one state was assumed to carry through to the next automatically, and it was the responsibility of each "operator" (the description of an action) to tell how to construct a new model. The form of the states themselves was not dictated by GPS and varied from one domain to another.

The resolution-based problem solvers (e.g., [Green, 1969; Waldinger and Lee, 1969]) maintained the GPS convention that every action was assumed capable of destroying any relation: in other words it was necessary to state explicitly such observations as that turning on a light switch does not alter the location of any of the objects in a room. To supply a large number of these facts (often called "frame axioms") was tedious, and they tended to distract the problem solver as well. Since most actions leave most of the world unchanged, we want our representation of the world to be biased to expect actions not to affect most existing relations. For a number of reasons we demand that these "obvious" facts be submerged in the representation, so that we (and our system) can focus our attention on the important things, the things that change.

The STRIPS problem solver (Fikes and Nilsson, 1971) was introduced to overcome these obstacles. In order to eliminate the frame axioms, STRIPS adopted the assumption that a given relation is left unchanged by an action unless it is explicitly mentioned in the "addlist" or the "deletelist" of the action: relations in the addlist are always true after the action is performed, while relations in the deletelist are not assumed to be true afterwards even if they were true before. Thus the frame axioms are assumed implictly for every action and relation unless the relation is included in the addlist or deletelist of the action. For instance, a (robot) action "go from A to B" might have "the robot is at B" in its addlist and "the robot is at A" in its deletelist. A relation such as "box C is in room 1" would be assumed to be unaffected by the action because it is not mentioned in either the addlist or the deletelist of the operator.

Henceforth, we shall refer to the belief that an action leaves all the relations in the model unchanged, unless specified otherwise, as the "STRIPS assumption."

A STRIPS model of a world situation, like a STRIPS operator, consists of an addlist and a deletelist: the addlist contains those relations that are true in the

corresponding situation but that may not have been true in the initial situation, and the deletelist contains those relations that may not be true in the corresponding situation even though they were true initially. Thus one can determine which relations are true, given the current model and the intial list of relations. Also, given a model and an operator, it is easy to apply the operator to the model and derive a new model. The STRIPS scheme keeps a complete record of all the past states of the system, while allowing the various models to share quite a bit of structure.

STRIPS operators are appealingly simple. In the next section we will examine how the sorts of techniques we have discussed apply if the actions are all STRIPS operators.

2.2 Regression and STRIPS operators

Suppose an action is represented as a STRIPS operator, and that the members of the addlist and the deletelist are all atomic—that is, they contain no logical connectives or quantifiers. It is singularly simple to pass a relation back over such an operator, because the interaction between the operator and the relation are completely specified by the addlist and the deletelist. In order for a relation to be true after the application of such an operator, it must (1) belong to the addlist of the operator, or else (2) be true before application of the operator and not belong to the deletelist of the operator. Thus the rule for passing any relation back over such a STRIPS operator is implicit in the operator description itself.

For instance, an operator such as "move A from B to C" might have addlist "A is on C" and "B is clear" and deletelist "A is on B" and "C is clear." Thus, when passed back over this operator, the relation "A is on C" becomes true, "A is on B" becomes false, and "C is on D" remains the same. The simplicity of regression in this case indicates that we should express our actions in this form whenever possible.

The problem-solver WARPLAN (Warren, 1974) uses precisely the same sort of skeleton model as we do, and uses an identical strategy for handling simultaneous goals, but restricts itself to an atomic add-deletelist representation for operators, thus achieving a marvelous simplicity. Although we imagine that WARPLAN would require extension before it could handle the sort problem or the interchanging of variable values, the principles involved in the WARPLAN design are a special case of those given here.

Thus the clarity of actions expressed in this form makes reasoning about them exceedingly easy. However, many have found the add-deletelist format for representing actions too restrictive. With the advent of the "artificial intelligence programming languages," it became more fashionable to represent actions "procedurally" so that the system designer could describe the effects of the action using the full power of a programming language. We shall examine the impact of the STRIPS assumption on some of these systems in the next section.

2.3 The use of contexts to represent a changing world

What is past, even the fool knows.
Homer, *Iliad*

The new AI languages include PLANNER (Hewitt, 1972), QA4 (Rulifson, *et al.*, 1972). CONNIVER (McDermott and Sussman, 1972) and QLISP (Wilber, 1976), a variant of QA4. A comparative survey of these languages is provided in (Bobrow and Raphael, 1974). Implementers of problem solvers in these languages are fond of saying their systems represent actions "procedurally," as computer programs, rather than "declaratively," as axioms or add-delete lists. Yet in each of these systems the STRIPS assumption is firmly embedded, and the procedures attempt to maintain an updated model by deleting some relations and adding others; which relations an action adds or deletes depends on a computation instead of being explicitly listed beforehand. The STRIPS assumption is expressed not procedurally or declaratively but structurally: it is built into the choice of representation. The more primitive systems (e.g., [Winograd, 1971; Buchanan and Luckham, 1974]), implemented in an early version of PLANNER, maintained a single model which they updated by adding and deleting relations.* This scheme made it impossible for the system to recall any but the most recent world situation without "backtracking," passing control back to an earlier state and effectively undoing any intermediate side effects. The more recent trend† has been to incorporate the assumption by a particular use of the "context" mechanism of the newer implementation languages. We must now describe the context mechanism and its use in building what we will call an "archeological model."

The context mechanism in QA4, CONNIVER, QLISP, AP/1, and HBASE operates roughly as follows: Each of these systems has a data base; assertions can be made and subsequently retrieved. Assertions and queries in these systems are always made with respect to an implicit or explicit context. If T_1 is a context, and we assert that B is on C with respect to T_1, the system will store that fact and answer accordingly to queries made with respect to T_1. There is an operation known as "pushing" a context that produces a new context, an immediate "descendant" of the original "parent" context. We may push T_1 any number of times, each time getting a new immediate descendant of T_1. If T_2 is a descendant of T_1, any assertion made with respect to T_1 will be available to queries made with respect to T_2.

---

*We do not mean to imply that all these systems were copying STRIPS; Winograd's work was done at the same time.

†See, for example, (Derksen, *et al.*, 1972; Sussman, 1973; Fahlman, 1974; McDermott, 1974; Fikes, 1975). (Balzer, *et al.*, 1974 and Tate, 1974) use the context mechanism of the AP/1 programming system and the HBASE data base system (Barrow, 1974), respectively, in exactly the same way.

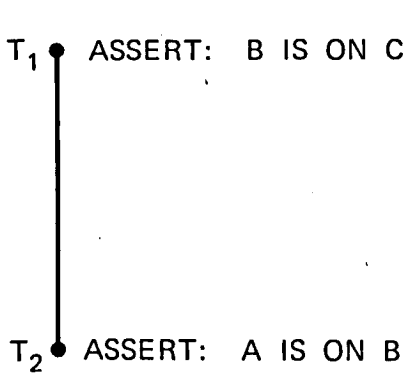


FIGURE 27

Thus if we ask whether B is on C with respect to T_2, we will be told "Yes" (in some fashion). However, assertions made with respect to that descendant are "invisible" to queries made with respect to its parent or any other context aside from its own descendants. For instance, if A is asserted to be on B with respect to T_2, that information will not be available to queries made with respect to T_1 (see Figure 27).

It is also possible to "delete" a relation with respect to a given context. If I delete the fact that B is on C with respect to T_2, the system will be unable to determine whether B is on C with respect to T_2 (on any of its descendants), but it will still know that B is on C with respect to T_1:

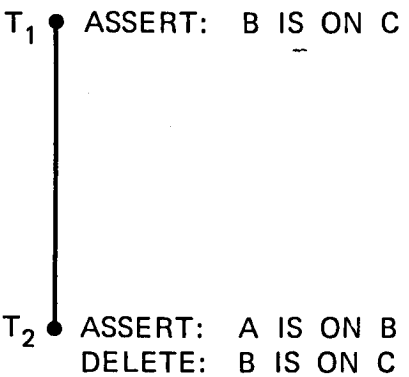


FIGURE 28

The convention taken in planning systems implemented in languages with such a "context-structured data base" has been to equate each situation with a context. Furthermore, if some action occurs in a given situation T_1, resulting in a new situation, the usual practice has been to equate the new situation with an immediate descendant T_2 of the given context T_1. Any relations that are produced by the action are asserted with respect to T_2; any relations that may be

disturbed by the action are deleted with respect to T_2. Other relations are still accessible in the new context. Thus if we are in situation T_1 and move block A onto block B from on top of block C, we construct a descendant T_2, asserting that A is on B and deleting that A is on C with respect to T_2. If B was known to be on block D in situation T_1, that information will still be available in situation T_2.

If T_2 is succeeded by another situation T_3, T_3 will be represented by a descendant of T_2, and so on. The structure of the sequence of contexts is represented as

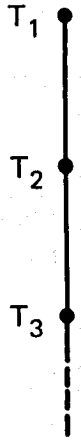


FIGURE 29

Each context is a descendant of the preceding context.

We will call this representation of the world an "archeological" model because it allows us to dig into successive layers of context in order to uncover the past.

In the balance of this paper we will propose that the archeological model is not always ideal. Because any assertion true in a context is automatically true in its descendants (unless specifically deleted), the use of archaeological models implicitly incorporates the STRIPS assumption, and accepts the STRIPS solution to the frame problem. Therefore, most of the planning systems implemented in the new AI languages use representations like that of STRIPS. We have been paying so much attention to the STRIPS assumption for the following reason: we are about to argue that in the future we may not want this assumption so firmly implanted in the structure of our problem solvers; indeed, some researchers have already begun to feel its constriction.

2.4 Influential actions

For want of a nail the shoe was lost,
For want of a shoe the horse was lost,
For want of a horse the rider was lost,
For want of a rider the battle was lost,
For want of a battle the kingdom was lost,
And all for the want of a horseshoe nail. Nursery Rhyme

The STRIPS assumption, embedded in the archeological model, has been so universally adopted because it banishes the frame axiom nightmare: it is no longer necessary to mention when an action leaves a relation unaffected because every action is assumed to leave every relation unaffected unless explictly stated otherwise. The assumption reflects our intuition about the world, and the archeological model represents the assumption in an efficient way. Having found a mechanism that rids us of the headaches of previous generations of artificial intelligence researchers, shouldn't we swear to honor and cherish it forever?

Indeed, so much can be done within the STRIPS-archeological model framework, and so great are the advantages of staying within its boundaries, that we only abandon it with the greatest reluctance. If we were only modelling robot acts we might still be content to update our models by deleting some relations and adding others. The death blow to this approach is dealt by programming language instructions such as the assignment statement.

Suppose we attempt to express an assignment statement $X \leftarrow Y$ by updating an archeological model. We must delete any relation of form P(X); furthermore, for every relation of form P(Y) in the model we must add a relation of form P(X). In addition, we may need to delete a relation of form "there is a z such that z has value b" even though it does not mention X explicitly. We may need to examine each relation in the model in order to determine whether it depends on X maintaining its old value. The consequences of this instruction on a model are so drastic and far reaching that we cannot afford to delete all the relations that the statement has made false.

How are we to represent the effects of an instruction such as sort2(X Y) on a model? If P(X Y) is the conjunction of everything that is known about X or Y, we might delete P(X Y) and assert $X \leqslant Y$ and $\neg$(P(X Y) or P(Y X)). This is a massive and unworkable formula if P(X Y) is at all complex; furthermore, it does not express our intuition about the sort, that whether P(X Y) or P(Y X) holds depends on whether or not X was less than or equal to Y before the sort took place. Knowledge of the previous relation between X and Y has been lost.*

Even in the robot domain, for which the STRIPS formalism was orginated, the archeological representation becomes awkward when considering actions with indirect side effects. For example, if a robot is permitted to push more than once box at a time, an operation such as "move box A to point x" can influence the locations of boxes B, C, and D.

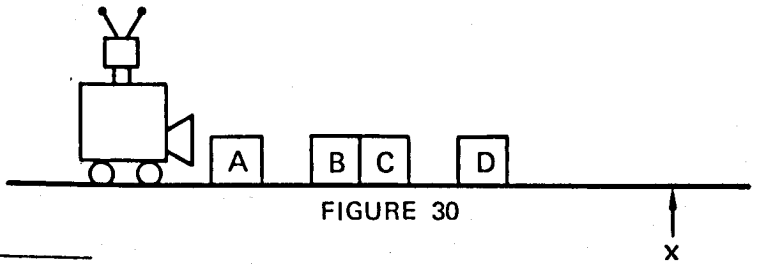


FIGURE 30

*A reply to some of this criticism appears in (Warren, 1976).

This situation becomes worse as the number of elements in the world increases: in moving a complex subassembly of a piece of equipment, we must change the location of every component of the subassembly. If we turn a subassembly upside down, we must replace every relation of form "x is on y" by the relation "y is on x," if x and y are components of the inverted assembly.

These actions are clumsy to model archeologically because so many relations need to be added and deleted from the model, and these relations may involve objects that are not explicitly mentioned by the operator. Furthermore, the operators are insensitive to whether or not these relations are relevant to the problem being solved.

Many of the more recent planning and modelling systems have been attempting to represent these "influential" actions, and we will soon examine how they have overcome the above obstacles. First let us point out that regression provides one technique for modelling these actions; for instance, we need not determine the location of any component indirectly affected by an action until a query concerning that component arises: thus, though many components may be moved, the system need only be concerned with a few of them. When a query about the location does arise, the regression technique will allow the new location to be determined from the original location and from the sequence of actions that has been performed on the subassembly. In particular, if the robot in the previous example (Figure 30) has moved the stack 10 feet to the right in moving box A to point x, the new location of box C will also be several feet to the right of the old location: of course, there is no need to compute the new location of C unless that information is requested.

We have seen that archeological models embed the STRIPS assumption; however, many of the more recent planning systems, while retaining the archeological structure, have been attempting to model actions that must be classified as influential. We will see in the next section how they have resolved the discrepancy.

2.5 Escaping from the STRIPS assumption

Once the archeological model was adopted, the designers of problem solvers devised mechanisms to loosen the STRIPS assumption embedded in their choice of representation.

Fahlman (Fahlman, 1974), using CONNIVER, wanted to simulate a robot that could lift and transport an entire stack or assembly of blocks in one step by carefully raising and moving the bottom block. We characterize this action as "influential" because many blocks will have their location changed when the bottom block is moved. Aware of the difficulty of maintaining a completely updated model, Fahlman distinguishes between "primary" and "secondary" relations. Primary relationships, such as the locations of the blocks, are fundamental to the description of the scene: an updated model is kept of all primary relationships. Secondary relationships, such as whether or not two blocks are touching, are defined in terms of the primary relationships and therefore can be deduced from the model, and added to it, only as needed. The system has

thereby avoided deducing large quantities of irrelevant, redundant secondary relationships.

Notice, however, that keeping an updated model of just the primary relationships may still be a sizable chore: for instance, at any moment the system must know the location of every block in the model, even though these locations are often themselves redundant; when a large subassembly is moved, the locations of each of the blocks in the subassembly can be computed from the location of the subassembly itself.

Furthermore, in Fahlman's system if a primary relationship is changed, all the secondary relationships that have been derived from that primary relationship and added to the model must be deleted at once to avoid potential inconsistency.

The modelling system of the SRI Computer Based Consultant (Fikes, 1975), implemented in QLISP, distinguishes between derived and explicitly asserted relations for the same reason that Fahlman distinguishes between primary and secondary data. However, in the SRI system the same relation might be derived in one instance and explictly asserted in another. Thus the location of a component could very well be derived from the location of a subassembly.

Like the Fahlman system, the SRI system deletes all the information derived from an assertion when it deletes the assertion itself.

Note that the SRI system does not behave at all well if the user tries to assert a complex relationship explicitly, say in a problem description. For instance, suppose the user says that block B is between blocks A and C. If the system then moves block A, it will still report that B is between A and C, because that relationship was explicitly asserted and not derived: the system has no way of knowing that it depends on the location of A.

The Fahlman system avoids this difficulty only by forbidding the user to assert any secondary relationships.

Both the Fikes and the Fahlman systems have the following scheme: define actions in terms of the important relationships that they modify, and then define the lesser relationships in terms of the important relationships. This simplifies the description of actions, makes model updating more efficient, and allows the system designer to introduce new relationships without needing to modify the actions' descriptions.

However, it may be impossible to define some lesser relationships in terms of the important ones; we may need to know directly how the lesser relationships are affected by actions. The moving of subassemblies provides a convenient example of this phenomenon.

Consider a row of blocks on a table.

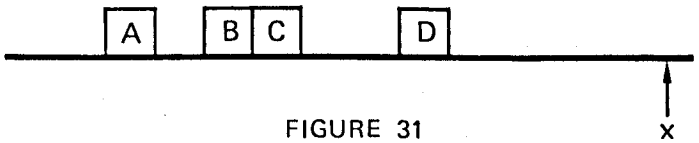


FIGURE 31

We want to move A several feet to the right, to point x. We can either slide A or lift it. If we lift it, blocks B, C, and D will stay where they are, whereas if we slide it, we will inadvertently carry the others along. It is expensive to expect the slide operator to update the model to include the new locations of all the blocks it affects: there may be many of these intermediate blocks and they may not be important to the problem being solved. On the other hand, we cannot expect an archeological system to deduce the new location of B from the new model in case that information turns out to be needed: in order to compute the location of B, the system needs to know whether A has been lifted or slid, and that information is not part of a conventional model. Thus, in an archeological model, locations of intermediate blocks must always be computed at the time the slide is added to the plan.

If skeleton models are adopted, on the other hand, the actions in the plan form an integral part of the model. If A is slid to x, only the new location of A would be explicitly included in the new model. If subsequently we need to determine the location of B, a regression rule sees that A has been slid and asks whether B is in the path of the slide; if not, the location of B after the slide is the same as before; otherwise, the new location of B is somewhere to the right of A.

In both the archeological and the skeletal representations, knowledge about the side effects of sliding must be explicitly expressed. In the skeleton model, the new locations of the intermediate blocks need not be computed until they are needed.

In archeological modelling, the description of an action must be expressed completely in a single operator. For an action with many side effects, the operator is likely to be a rather large and opaque program. Skeleton modelling does not eliminate the need to describe the effects of an action explicitly; however, it does allow the description to be spread over many smaller, and usually clearer programs. Furthermore, one can alter a system to handle new relations merely by adding new regression rules, without changing any previously defined operators. In short, skeleton modelling can sometimes make a system more transparent and modular, as well as more efficient.

Skeleton models do not discard the STRIPS assumption. If this assumption were abandoned, the frame problem would be back upon us at once: for every relation and action it would be necessary to state or deduce a regression rule whether or not the action had any effect at all on the relation. Instead, skeleton models contain a default rule stating that if no other regression rule applies, a given relation is assumed to be left unchanged by a given action. This rule states the STRIPS assumption precisely but does not freeze it into a structure. We have lost in efficiency if actions really do have few side effects, because the archeological model does embed the STRIPS assumption in a structural way and requires no computation if it applies, whereas a skeleton model can only apply the assumption after all the regression rules have failed. The extent to which this modelling technique will be economic depends entirely on the "influence" of actions of the plan–the degree to which they affect the relations in the model.

If skeleton models are adopted, the context mechanism need not be dropped altogether as a way of representing distinct world situations; however, descendent contexts cannot be used to represent successive world states. Our implementation of skeleton models uses contexts in a different way, which we will outline in the next section.

2.6 The use of contexts to implement skeleton models

Recall that we can "push" a given context any number of times, creating a new immediate descendant with every push. These new contexts are independent from each other—none of them is descended from any of the others, and an assertion made with respect to one of them will be invisible to the rest.

In our implementation of skeleton models we represent each situation by a context, but successive situations are all immediate descendents of a single global context T. Thus if situation T_2 results from situation T_1 by performing some act, T_1 and T_2 will both be immediate descendants of T, created by pushing T; T_2 will not be a descendant of T_1. We can represent the skeleton model context structure as follows:

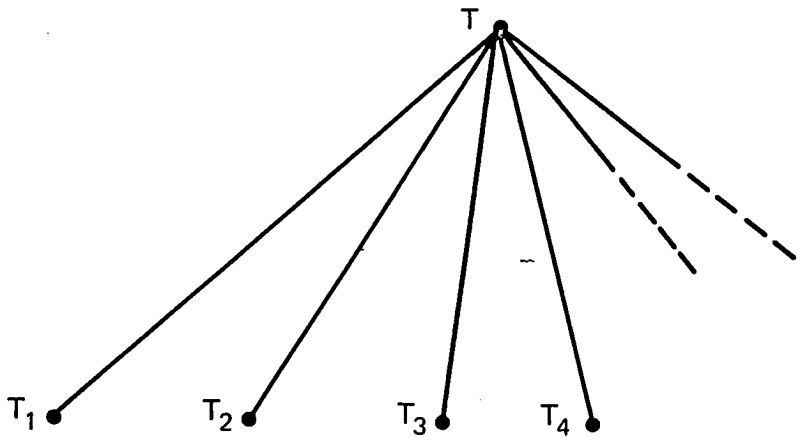


FIGURE 32

Asserting a relation with respect to T_1 does not automatically make it true with respect to T_2, and so on. The only relations asserted in the global context T are the eternal verities.

Since the structure of the skeleton model does not imply any relationship at all between successive states, we represent such knowledge procedurally, by the regression rules for passing a relation back from one state to the preceding one. We suffer a possible loss of efficiency in abandoning the archeological model, but we gain in flexibility and in our ability to represent influential operators efficiently. We do not need to struggle against the assumption incorporated into our representation.

Of course, it is possible to implement skeleton models without using a context mechanism. Problem solvers of the sort advocated by Kowalski (Kowalski, 1974) or implemented by Warren (Warren, 1974) embed a skeleton model representation in a predicate logic formalism in which states of the world are represented by explicit state variables, just as in the early theorem-proving approach. These systems are especially elegant in that the regression rules are indistinguishable from the operator descriptions. They both accept the STRIPS add-deletelist operator representation, but we can envision their incorporating the sort of regression we have employed without requiring any fundamental changes in structure. Hewitt (Hewitt, 1975) has indicated that a version of what we have called skeleton modelling has also been developed independently in the actor formalism, and Sacerdoti (Sacerdoti, 1975) uses another version in conjunction with the procedural net approach.

2.7 Hypothetical worlds

What might have been is an abstraction
Remaining a perpetual possibility
Only in a world of speculation.
What might have been and what has been
Point to one end, which is always present.
Footfalls echo in the memory
Down the passage which we did not take
Towards the door we never opened
Into the rose-garden.

T.S. Eliot, *Four Quartets*

Although so far we have avoided discussing the formation of conditional plans in this paper, it may now be useful to note that using descendent contexts to split into alternate hypothetical worlds (cf. [Rulifson, *et al.*, 1972; McDermott, 1974; Manna and Waldinger, 1974]) is entirely consistent with using independent contexts in skeleton models, but presents something of a problem to archeological models.

In both archeological and skeletal models it is common to represent hypothetical worlds by descendent contexts. For instance, to prepare alternate plans depending on whether or not it is raining in a situation represented by context T_1, two new contexts T_1' and T_1'' are formed, corresponding to the cases in which it is raining and it is not raining, respectively. T_1' and T_1'' are both descendants of T_1, so that any relations known in T_1 will automatically be assumed about T_1' and T_1'' also, as one would have hoped. Furthermore, in T_1' it is asserted to be raining, while in T_1'' it is asserted not to be raining.

The plan for the rainy case would be represented as a sequence of contexts that follows T_1'. In an archeological model these would be successive descendants of T_1' (Figure 33), while in a skeleton model these would be independent contexts linked by regression rules. A similar sequence of contexts beginning with T_1'' would correspond to the plan for the case in which it is not rainy.

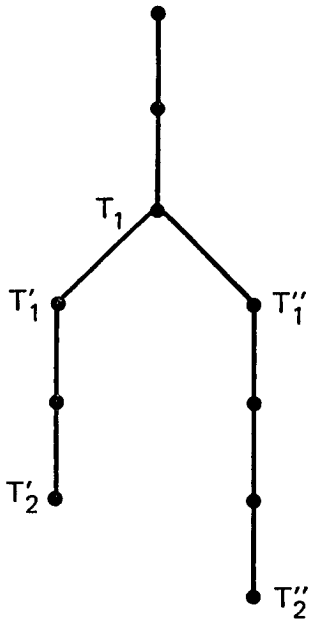


FIGURE 33

Eventually we may reach a situation T'_2 and T''_2 in each plan, respectively, after which it becomes irrelevant whether or not it was raining in T_1. In other words our ultimate goal may now be achieved by a single plan that will work in either T'_2 or T''_2. Therefore we would like to join our two plans back together into a single plan; we want to form a new context T_2 such that P is true in T_2 if and only if it is true in both T'_2 and T''_2. This can be done in a skeleton model by creating an independent context T_2 linked to the previous contexts by the following regression rule: to establish R in T_2, establish R in both T'_2 and T''_2.

The situation becomes more difficult if one attempts to maintain an updated archeological model. One could take the following approach: if P and Q are the conjunction of all that is known in T'_2 and T''_2, respectively, then assert (P or Q) with respect to T_2. However, (P or Q) is likely to be an unwieldy formula, and we may have lost the information that P corresponds to the rainy situation and Q to the nonrainy one.

We regret that our treatment of hypothetical situations is so terse. A discussion of our own approach, with examples, is given in (Manna and Waldinger, 1974).

2.8 Complexity

> Home is where one starts from. As we grow older
> The world becomes stranger, the pattern more complicated
> Of dead and living.
>
> T.S. Eliot, *Four Quartets*

Perhaps we should say a few words contrasting the work reported here with recent work of Sussman (Sussman, 1973) and Sacerdoti (Sacerdoti, 1975). Although both of these works deal in some of their aspects with simultaneous goals, the principal thrust of their interests is different from ours, and so comparisons are likely to be shallow.

Sussman's main interest is the acquisition of knowledge. Thus he wants his system to learn how to handle simultaneous goals, and is more concerned with learning than with simultaneous goals themselves. We, on the other hand, want our system to know how to handle simultaneous goals from the start, and are not (at present) concerned with learning at all.

The sort of program modification we do is distinct from debugging: the program we are modifying correctly achieves one goal, and we want it to achieve another. We also refrain from actually executing our programs, and ultimately produce programs that are guaranteed correct, whereas Sussman produces programs that may have undiscovered bugs. It is plausible that in tackling more complex problems we will want to introduce bugs and later correct them. We imagine this happening in problems involving serveral levels of detail: a program may work correctly in a crude way, but still contain many minor errors. The problems we have been considering are simple enough so that we have not been forced into using these techniques.

Similarly we view Sacerdoti's procedural nets, like his earlier abstraction hierarchies (Sacerdoti, 1974) as a way of dealing with complexity by submerging detail until a grossly correct plan has been developed. Then the plan is examined in greater depth, and difficulties are ironed out as they emerge. The Sacerdoti formalism can easily represent actions with many subsidiary side effects: these effects are considered only after the initial (approximate) plan has been formulated.

In approaching several simultaneous goals, Sacerdoti develops plans to achieve each of the goals separately; as interactions between the plans are observed, the system will impose orderings on the steps ("Step F_i from plan F must be executed before step G_j from plan G") and even alter the plans themselves to make them impervious to the effects of the other plans. Actions are represented essentially by addlists and deletelists, and the "critics" (cf. [Sussman, 1973]) that recognize the interactions between plans rely strongly on this representation, although the critic principle is more general.

Sacerdoti's approach to simultaneous goals is partially dictated by his application: a consultant system advising a human amateur in a repair task. The user may choose to order the plan steps in any of a number of valid ways; the system cannot force an order except where that order is necessary to avoid harmful interactions; therefore it maintains a highly parallel plan whenever possible until the user himself has selected the order. In a sense, Sacerdoti's system must anticipate all possible plans to achieve a task.

Sacerdoti's idea, deciding what order in which to approach goals only after having done some planning for each of them, is intriguing and avoids a certain

amount of goal reordering. However, we believe we will not make best use of hierarchical planning until we are ready to wade into deeper waters of complexity.

2.9 Recapitulation

> You say I am repeating
> Something I have said before. I shall say it again.
>
> T.S. Eliot, *Four Quartets*

In this section we will briefly repeat the main points of the argument in Part 2.

The earliest problem solvers maintained entirely separate models corresponding to each state of the world. In GPS, each operator had the responsibility of constructing a completely new model, whereas in the resolution-based systems the description of the new model created by an action was distributed between several axioms, some describing how relationships were changed by the action, and others (the frame axioms) telling which relationships remained the same.

In an effort to do away with troublesome and obvious frame axioms, later problem solvers adopted what we have called the "STRIPS assumption," that any action will not change most relations, and therefore they described an action by telling which relations it adds and which relations it deletes from the model. The "addlists" and "deletelists" were either given explicitly or computed. Any relation not explicitly added or deleted by an action was assumed to be unaffected.

Systems implemented in artificial intelligence programming languages having a "context" feature tended to incorporate the STRIPS assumption by equating states of the world with contexts, and representing states that occur after a given state by successive descendants of the given context; since any relation asserted with respect to the given context is considered to be true with respect to any of its descendents unless explicitly deleted, the STRIPS assumption is expressed structurally in this "archeological" representation.

Meanwhile, the designers of problem-solving systems entered domains in which the STRIPS assumption began to break down: areas in which the world was modelled in such detail, or in which objects were so highly interrelated, that actions might have many consequences, most of which were irrelevant to the problem at hand. The STRIPS assumption and the archeological structure that expresses it become an obstacle here: it would be cumbersome and inefficient for the description of the action to have to make all these changes in the model. Recent problem solvers have attempted to escape from the STRIPS assumption by distinguishing between the important relations, which are always updated in the model, and the lesser relations, which are defined in terms of the important relations and which are only updated as necessary. These measures are inadequate largely because the designer of the system is prevented from stating

explicitly how the lesser relationships are affected by the various actions.

The regression technique advocated here and elsewhere provides a method whereby the actions in the plan become an important part of the model, from which a relational description of the world can be "fleshed out" as necessary. The context mechanism can be used to represent this type of "skeleton model," but successive states are represented as parallel contexts instead of descendants. This latter representation has the additional advantage of being consistent with the use of descendent contexts to represent hypothetical worlds, and with the program modification technique introduced in Part 1.

In my end is my beginning.
T.S. Eliot, *Four Quartets*

ACKNOWLEDGMENTS

This work has been developed through discussions with Zohar Manna and Earl Sacerdoti. The manuscript has benefited from the comments of Rich Fikes, Earl Sacerdoti, and Bert Raphael. The ideas presented here have also been influenced by conversations with Mike Wilber, Bob Boyer, Nachum Dershowitz, Rod Burstall, John Darlington, Gordon Plotkin, Bob Kowalski, Alan Bundy, Bernie Elspas, Nils Nilsson, Peter Hart, Ben Wegbreit, Carl Hewitt, Harry Barrow, Cordell Green, Avra Cohn, Dave Barstow, Doug Lenat, and Lou Steinberg. Mike Wilber has been of special assistance in the use of the QLISP system, which is based on INTERLISP. Our efforts have been encouraged by the environments provided by the Artificial Intelligence Center at SRI, the Department of Artificial Intelligence at the University of Edinburgh, and the Applied Mathematics Department at Weizmann Institute. Linda Katuna and Lorraine Staight prepared many versions of the manuscript.

The National Science Foundation Office of Computing Activities supported this work through Grant GJ-36146.

REFERENCES

Balzer, R.M., N.R. Greenfeld, M.J. Kay, W.C. Mann, W.R. Ryder, D. Wilczynski, and A.L. Zobrist (1974) Domain-independent automatic programming. *Information Processing 74: Proc IFIP 74,* 2, 326-330.

Barrow, H.G. (1974) HBASE. *POP-2 Library Documentation,* Department of Artificial Intelligence, University of Edinburgh, Edinburgh.

Bobrow, D.G. and B. Raphael (1974) New programming languages for artificial intelligence research. *ACM Computer Surveys,* 6, 3, 155-174.

Boyer, R.S. and JS Moore (1973) Proving theorems about LISP functions. *Proc IJCAI3,* 486-493, Stanford, CA, also in *JACM,* 22, 1, 129-144.

Buchanan, J.R. and D.C. Luckham (1974) On automating the construction of programs. *Informal Memo,* Artificial Intelligence Laboratory, Stanford University, Stanford, CA.

Darlington, J. (1975) Application of Program Transformation to Program Synthesis. *Proc Colloques IRIA:* Proving and improving programs, 133-144, Arc et Senans, France.

Darlington, J. and R.M. Burstall (1973) A system which automatically improves programs. *Proc IJCAI3,* 479-485, Stanford, CA.

Derksen, J., J.F. Rulifson and R.J. Waldinger (1972) The QA4 language applied to robot planning. *AFIPS* 41, Part II, 1181-1187.

Dijkstra, E.W. (1975) Guarded commands, non-determinacy and a calculus for the derivation of programs. *Proceedings, International Conference on Reliable Software* 2-2.13, Los Angeles, CA.

Fahlman, S. (1974) A planning system for robot construction tasks. *Artificial Intelligence*, 5, 1, 1-49.

Fikes, R.E. (1975) Deductive retrieval mechanisms for state description models. *Technical Note 106*, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA.

Fikes, R.E. and N.J. Nilsson (1971) STRIPS: A new approach to the application of theorem proving in problem solving, *Artificial Intelligence*, 2, 3/4, 189-208.

Floyd, R.W. (1967) Assigning meanings to programs. *Mathematical Aspects of Computer Science*, Proceedings of a Symposium on Applied Mathematics Vol. 19, American Mathematical Society, 19-32.

Goldstine, H.H. and J. von Neumann (1947) Planning and Coding Problems for an Electronic Computer Instrument. *Collected Works of John von Neumann* 5, 80-235 (Pergamon Press, New York, 1963).

Green, C.C. (1969) Application of theorem proving to problem solving. *Proc IJCAI* 219-239, Washington, DC.

Green, C.C., R.J. Waldinger, D.R. Barstow, R. Elschlager, D.B. Lenat, B.P. McCune, D.E. Shaw and L.I. Steinberg (1974) Progress report on program-understanding systems. *Memo AIM-240*, Stanford Artificial Intelligence Laboratory, Stanford University, Stanford, CA.

Green, C.C. and D. Barstow (1976) A hypothetical dialogue exhibiting a knowledge base for a program-understanding system. *Machine Intelligence* 8, (eds. Elcock, E.W. and Michie, D.), Ellis Horwood Ltd. and John Wiley.

Hewitt, C. (1972) Description and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. *Ph.D. Thesis* Massachusetts Institute of Technology, Cambridge, Mass.

Hewitt, C. (1975) How to use what you know. *Proc IJCAI4*, 189-198, Tbilisi, Georgia, USSR.

Hoare, C.A.R. (1969) An axiomatic basis for computer programming. *CACM*, 12, 10, 576-580, 583.

Katz, S.M. and Z. Manna (1976) Logical analysis of programs, *CACM* 19, 4, 188-206.

King, J.C. (1969) A Program Verifier. *Ph.D. Thesis*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Kowalski, R. (1974) Logic for Problem Solving. *Memo No. 75*, Department of Computational Logic, University of Edinburgh, Edinburgh.

Manna, Z. (1968) Termination of Algorithms. *Ph.D. Thesis* Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Manna, Z. (1974) *Mathematical Theory of Computation* McGraw Hill, New York.

Manna, Z. and R.J. Waldinger (1974) Knowledge and reasoning in program synthesis. *Artificial Intelligence* 6 2, 175-208.
CA.

McCarthy, J. and P. Hayes (1969) Some philosophical problems from the standpoint of artificial intelligence, *Machine Intelligence* 4, (eds. Meltzer, B. and Michie, D.), American Elsevier, New York.

McDermott, D.V. (1974) Assimilation of new information by a natural language-understanding system, *AI Memo 291*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

McDermott, D.V. and G.J. Sussman (1972) The Conniver Reference Manual. *AI Memo 259* (revised 1973), Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Moore, JS (1975) Introducing iteration into the pure LISP theorem prover. *IEEE Transactions on Software Engineering*, SE-1, 3, 328-338.

Naur, P. (1966) Proof of algorithms by general snapshots. *BIT* 6, 4, 310-316.

Newell, A., J.C. Shaw, and H.A. Simon (1960) Report of a General Problem-Solving Program for a Computer. Information Processing. *Proceedings of an International Con-*

ference on Information Processing, 256-264. UNESCO, Paris, France.

Rulifson, J.F., J.A.C. Derksen, and R.J. Waldinger (1972) QA4: A procedural calculus for intuitive reasoning. *Technical Note 73*, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA.

Sacerdoti, E.D. (1974) Planning in a Hierarchy of Abstraction Spaces *Artificial Intelligence*, 5, 2, 115-135.

Sacerdoti, E.D. (1975) The non-linear nature of plans. *Proc IJCAI4*, 206-214, Tbilisi, Georgia, USSR.

Sussman, G.J. (1973) A Computational Model of Skill Acquisition. *Ph.D. Thesis*. Massachusetts Institute of Technology, Cambridge, MA.

Tate, A. (1974) INTERPLAN: A plan generation system that can deal with interactions between goals. *Memorandum MIP-R-109*, Machine Intelligence Research Unit, University of Edinburgh, Edinburgh.

Turing, A.M. (1950) Checking a large routine. *Report of a Conference on High Speed Automatic Calculating-Machines*, 66-69. University of Toronto, Toronto.

Waldinger, R.J. and R.C.T. Lee (1969) PROW: A step toward automatic program writing. *Proc IJCAI*, 241-252, Washington, D.C.

Waldinger, R.J. and K.N. Levitt (1974) Reasoning about programs. *Artificial Intelligence*, 5. 3. 235-316.

Warren, D.H.D. (1976) Generating Conditional Plans and Programs. *Proc AISB Summer Conference*, 344-354, Edinburgh.

Wegbreit, B. (1974) The synthesis of loop predicates, *CACM* 17, 2, 102-112.

Wilber, M. A QLISP Reference Manual. *Technical Note 118*, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA.

Winograd, T. (1971) Procedures as a Representation for Data in a Computer Program for Understanding Natural Language. *Ph.D. Thesis*, Massachusetts Institute of Technology, Cambridge, MA. also appears as *Understanding Natural Language* Academic Press, New York, NY.

Winograd, T., (1974) Artificial Intelligence–when will computers understand people? *Psychology Today*, 7, 12, 73-79.

MEASUREMENT OF KNOWLEDGE

7

Dynamic Probability, Computer Chess, and the Measurement of Knowledge*

I. J. Good

Virginia Polytechnic Institute and State University
Blacksburg, Virginia

Philosophers and "pseudognosticians" (the artificial intelligentsia[1]) are coming more and more to recognize that they share common ground and that each can learn from the other. This has been generally recognized for many years as far as symbolic logic is concerned, but less so in relation to the foundations of probability. In this essay I hope to convince the pseudognostician that the philosophy of probability is relevant to his work. One aspect that I could have discussed would have been probabilistic causality (Good, 1961/62), in view of Hans Berliner's forthcoming paper "Inferring causality in tactical analysis", but my topic here will be mainly dynamic probability.

The close relationship between philosophy and pseudognostics is easily understood, for philosophers often try to express as clearly as they can how people make judgments. To parody Wittgenstein, what can be said at all can be said clearly *and it can be programmed.*

A paradox might seem to arise. Formal systems, such as those used in mathematics, logic, and computer programming, can lead to deductions outside the system only when there is an input of assumptions. For example, no probability can be numerically inferred from the axioms of probability unless some probabilities are assumed without using the axioms: *ex nihilo nihil fit.*[2] This leads to the main controversies in the foundations of statistics: the controversies of whether intuitive probability[3] should be used in statistics and, if so, whether it should be logical probability (credibility) or subjective (personal). We who talk about the probabilities of hypotheses, or at least the relative probabilities of pairs of hypotheses (Good, 1950,1975) are obliged to use intuitive probabilities. It is difficult or impossible to lay down precise rules for specifying the numerical values of these probabilities, so some of us emphasize the need for subjectivity, bridled by axioms. At least one of us is convinced, and has repeatedly emphasized for the last thirty years, that a subjective probability can usually be judged only to lie in some interval of values, rather than having a

*Reprinted from *Fīrbush News*, with the permission of the editor.

sharp numerical value (Good, 1950). This approach arose as a combination of those of Keynes (Keynes, 1921) and of Ramsey (Ramsey, 1931); and Smith's (Smith, 1961) proof of its validity based on certain desiderata, was analogous to the work of Savage (Savage, 1954) who used sharp probabilities.

It is unfortunately necessary once again to express this theory of "comparative subjective probability" in a little more detail before describing the notion of dynamic probability. The theory can be described as a "black box" theory, and the person using the black box is called "you." The black box is a formal system that incorporates the axioms of the subject. Its input consists of your collection of judgments, many of which are of the form that one probability is not less than another one, and the output consists of similar inequalities better called "discernments." The collection of input judgments is your initial *body of beliefs, B*, but the output can be led back into the input, so that the body of beliefs grows larger as time elapses. The purpose of the theory is to enlarge the body of beliefs and to detect inconsistencies in it. It then becomes your responsibility to resolve the inconsistencies by means of more mature judgment. The same black box theory can be used when utilities are introduced and it is then a theory of rationality (Good, 1950,1952).

This theory is not restricted to rationality but is put forward as a model of *all* completed scientific theories.

It will already be understood that the black box theory involves a time element; but, for the sake of simplicity in many applications, the fiction is adopted (implicitly or explicitly) that an entirely consistent body of beliefs has already been attained. In fact one of the most popular derivations of the axioms of probability is based on the assumption that the body of beliefs, including judgments of "utilities" as well as probabilities, is consistent.[4]

One advantage of assuming your body of beliefs to be consistent, in a static sense, is that it enables you to use conventional mathematical logic, but the assumption is not entirely realistic. This can be seen very clearly when the subject matter is mathematics itself. To take a trivial, but very clear example, it would make sense for betting purposes to regard the probability as 0.1 that the millionth digit of π is a 7, yet we know that the "true probability" is either 0 or 1. If the usual axioms of intuitive probability are assumed, together with conventional static logic, it is definitely inconsistent to call the probability 0.1. If we wish to avoid inconsistency we must change the axioms of probability or of logic. Instead of assuming the axiom that $P(E|H) = 1$ when H logically implies E, we must assume that $P(E|H) = 1$ when we have *seen* that H logically implies E. In other words probabilities can change in the light of calculations or of pure thought without any change in the *empirical* data (cf. Good, 1950, p. 49, where the example of chess was briefly mentioned). In the past I have called such probabilities "sliding," or "evolving," but I now prefer the expression *dynamic probability*.[5] It is difficult to see how a subjective probability, whether of a man or of a machine, can be anything other than a dynamic one. We use dynamic probability whenever we make judgments about the truth or falsity of mathematical theorems, and competent mathematicians do this frequently, though

usually only informally. There is a naive view that mathematics is concerned only with rigorous logic, a view that arises because finished mathematical proofs *are* more or less rigorous. But in the process of finding and conjecturing theorems every real mathematician is guided by his judgments of what is probably true.[6] This must have been known for centuries, and has been much emphasized and exemplified by Polya (Polya, 1941,1954). A good "heuristic" in problem solving is one that has a reasonable chance of working.[7]

Once the axioms of probability are changed, there is no genuine inconsistency. We don't have to say that P(E|H) has more than one value, for we can denote its value at time t by $P_t(E|H)$, or we can incorporate a notation for the body of beliefs B_t if preferred. There is an analogy with the FORTRAN notation, as in x = x + 3, where the symbol x changes its meaning during the course of the calculation without any real inconsistency.[8]

Believing, as I did (and still do), that a machine will ultimately be able to simulate all the intellectual activities of any man, if the machine is allowed to have the same mechanical help as the man,[9] it used to puzzle me how a machine could make probability judgments. I realized later that this is no more and no less puzzling than the same question posed for a man instead of a machine. We *ought* to be puzzled by how judgments are made, for when we know how they are made we don't call them judgments (Good, 1959B).[10] If judgments ever cease then there will be nothing left for philosophers to do. For philosophical applications of dynamic probability see Appendix A.

Although dynamic probability is implicitly used in most mathematical research it is even more clearly required in the game of chess.[11] For in most chess positions we cannot come as close to certainty as in mathematics. It could even be reasonably argued that the sole purpose of analyzing a chess position, in a game, is for the purpose of improving your estimate of the dynamic probabilities of winning, drawing, or losing. If analysis were free, it would pay you in expectation to go on analyzing until you were blue in the face, for it is known that free evidence is always of non-negative expected utility (for example, (Good, 1967A), but see also (Good, 1974)). But of course analysis is not free, for it costs effort, time on your chess clock, and possibly facial blueness. In deciding formally how much analysis to do, these costs will need to be quantified.

In the theory of games, as pioneered mainly by von Neumann (von Neumann, 1944/47), chess is described as a "game of perfect information," meaning that the rules involve no reference to dice and the like. But in practice most chess positions cannot be exhaustively analyzed by any human or any machine, *present or future.*[12] Therefore play must depend on probability even if the dependence is only implicit. Caissa is a cousin of the Moirai after all.

Against this it can be argued that the early proposals for helping humans and computers to play chess made use of evaluation functions (for quiescent positions) and did not rely on probability, dynamic or otherwise. For example, the beginner is told the value of the pieces, P = 1, B = 3.25, etc. and that central squares are usually more valuable than the others.[13] But an evaluation function can be fruitfully interpreted in probabilistic terms and we now recall a con-

jectured approximate relationship that has been proposed (Good, 1959B,1967B) by analogy with the technical definition of weight of evidence.

The weight of evidence, provided by observations E, in favour of one hypothesis H_1, as compared with another one H_2, is defined as

$$\log \frac{O(H_1/H_2|E)}{O(H_1/H_2)} = \log \frac{P(E|H_1)}{P(E|H_2)}$$

where P denotes probability and O denotes odds. In words, the weight of evidence, when added to the intial log-odds, gives the final log-odds. The expression "weight of evidence," in this sense, was used independently in (Pierce, 1878), (Good, 1950), and (Minsky and Selfridge, 1961). Weight of evidence has simple additive and other properties which make it, in my opinion, by far the best explicatum for corroboration (Good, 1960/68, 1968, 1975). The conjecture is that *ceteris paribus* the weight of evidence in favour of White's winning as compared with losing, in a given position, is roughly proportional to her advantage in material, or more generally to the value of her evaluation function, where the constant of proportionality will be larger for strong players than for weak ones. The initial log-odds should be defined in terms of the playing strengths of the antagonists, and on whether the position is far into the opening, middle-game, or end-game, etc. Of course this conjecture is susceptible to experimental verification or refutation or improvement by statistical means, though not easily; and at the same time the conjecture gives additional meaning to an evaluation function.[14] As an example, if an advantage of a pawn triples your odds of winning as compared with losing, then an advantage of a bishop should multiply your odds by about $3^{3.25} = 35.5$. This quantitative use of probability is not in the spirit of Polya's writings, even if interval estimates of the probabilities are used.

If dynamic probability is to be used with complete seriousness, then it must be combined with the principle of rationality (see Appendix A). First you should decide what your utilities are for winning, drawing, and losing, say u_W, u_D, and u_L. More precisely, you do not need all three parameters, but only the ratio $(u_W - u_D)/(u_D - u_L)$. Then you should aim to make the move, or one of the moves, that maximize the mathematical expectation of your utility, in other words you should aim to maximze

$$p_W u_W + p_D u_D + p_L u_L \qquad (1)$$

where p_W, p_D, and p_L are your dynamic probabilities of winning, drawing, or losing. When estimating (1) you have to allow for the state of the chess clock so that the "costs of calculation," mentioned in Appendix A, are very much in the mind of the match chess player.[15] This is not quite the whole picture because you might wish to preserve your energy for another game: this accounts for many "grandmaster draws."

Current chess programs all depend on tree analysis, with backtracking, and the truncation of the tree at certain positions. As emphasized in (Good, 1967B),

it will eventually be necessary for programs to handle descriptions of positions[16] if Grandmaster status is to be achieved, and the lessons derived from this work will of course change the world, but we do not treat this difficult matter in this paper.

For the moment let us suppose that the problem has been solved of choosing the nodes where the tree is to be truncated. At each such node the probabilities p_W, p_D, and p_L are a special kind of dynamic probability, namely *superficial* or *surface* probabilities, in the sense that they do not depend on an analysis in depth. The evaluation function used at the end-nodes, which is used for computing these three probabilities, might depend on much deep cogitation and statistical analysis, but this is not what is meant here by an "analysis in depth." Then the minimax backtracking procedure can be used; or expectimaxing if you wish to allow for the deficiencies of your opponent,[17] and for your own deficiencies. In this way you can arrive at values of the dynamic probabilities $p_W{}^0$, $p_D{}^0$, and $p_L{}^0$ corresponding to the positions that would arise after each of your plausible moves in the *current* position, π_0. Of course these probabilities depend on the truncation rules (pruning or pollarding).

Some programs truncate the analysis tree at a fixed depth but this is very unsatisfactory because such programs can never carry out a deep combination. Recognizing this, the earliest writers on chess programming, as well as those who discussed chess programming intelligently at least ten years earlier,[18] recognized that an important criterion for a chess position π to be regarded as an endpoint of an analysis tree was *quiescence*. A quiescent position can be defined as one where the player with the move is neither threatened with immediate loss, nor can threaten his opponent with immediate loss. The primary definition of "loss" here is in material terms, but other criteria should be introduced. For example, the advance of a passed pawn will often affect the evaluation non-negligibly. We can try to "materialize" this effect, for example, by regarding the value of a passed pawn, not easily stopped, as variable. My own proposals are 1¼ on the fourth rank, 1½ on the fifth rank, 3 on the sixth, 5 on the seventh, and 9 on the eighth!, but this is somewhat crude.[19]

An informal definition of a turbulent position is a combinative one. For example, the position: White K at c3, R at c8; Black K at a1, R at a4; is turbulent. But if Black also has a Q at f6, then White's game is hopeless, so the turbulence of the position does not make it much worth analyzing.

Hence in (Good, 1967B, p. 114) I introduced a term *agitation* to cover both turbulence and whether one of p_W, p_D, and p_L is close to 1. Apart from considering whether to threaten to take a piece, in some potential future position π, we should consider whether the win of this piece would matter much. Also, instead of considering one-move threats, it seems better to consider an analysis of unit cost, which might involve several moves, as, for example, when checking whether a pawn can be tackled before it touches down in an end-game. The definition of the *agitation* $A(\pi)$ was the expected value of $|U(\pi|\$) - U(\pi)|$ where $U(\pi)$ is the superficial utility of π and $U(\pi|\$)$ is the utility of π were a unit

of amount of analysis to be done. ($U(\pi|\$)$ is a subjective random variable before the analysis is done.)

But the depth from the present position π_0 to π is also relevant in the decision whether to truncate at π. More exactly, the dynamic probability $P(\pi|\pi_0)$ that position π will be reached from π_0 is more relevant than the depth. We could even reasonably define the probabilistic depth as proportional to $-\log P(\pi|\pi_0)$ and the effective depth of the whole analysis as $-\Sigma P(\pi|\pi_0) \log P(\pi|\pi_0)$, summed over all endpoints π, as suggested in (Good, 1967B). But the most natural criterion for whether to treat π as an endpoint in the analysis of π_0 is obtained by setting a threshold on $P(\pi|\pi_0)A(\pi)$. The discussion of agitation and allied matters is taken somewhat further in (Good, 1967B, pp. 114-115).

As a little exercise on dynamic probability let us consider the law of multiplication of advantage which states that "with best play on both sides we would expect the rate of increase of advantage to be some increasing function of the advantage." This might appear to contradict the conjecture that the values of the pieces are approximately proportional to weights of evidence in favour of winning rather than losing. For we must have the "Martingale property" $E(p_t|p_0) = p_0$, where p_0 and p_t are the probabilities of winning at times 0 and t. This only sounds paradoxical if we forget the elementary fact that the expectation of a function is not usually equal to that same function of the expectation. For example, we could have, for some $\epsilon > 0$,

$$E(\log \frac{p_t}{1-p_t}) \approx (1+\epsilon)^t \log \frac{p_0}{1-p_0} \qquad (2)$$

without contradicting the Martingale property, and (2) expresses a possible form of the law of multiplication of advantage, though it cannot be very accurate.

An idea closely associated with the way that dynamic probabilities can vary is the following method for trying to improve any given chess program. Let the program starting in a position π_0, play against itself, say for the next n moves, and then to quiescence, at say π_1. Then the odds of winning from position π_1, or the expected utility, could be used for deciding whether the plan and the move adopted in position π_0 turned out well or badly. This information could be used sometimes to change the decision, for example, to eliminate the move chosen before revising the analysis of π_0. This is not the same as a tree analysis alone, starting from π_0, because the tree analysis will often not reach the position π_1. Rather, it is a kind of learning by experience. In this procedure n should not be at all large because non-optimal moves would introduce more noise the larger n was taken. The better the standard of play the larger n could be taken. If the program contained random choices, the decision at π_0 could be made to depend on a *collection* of sub-games instead of just one. This idea is essentially what humans use when they claim that some opening line "appears good in master practice."

To conclude this paper I should like to indicate the relevance of dynamic probability to the quantification of knowledge, for which Michie proposed a non-probabilistic measure.[20] As he points out, to know that $12^3 = 1728$ can be

better than having to calculate it, better in the sense that it saves time. His discussion was non-probabilistic so it could be said to depend, at least implicitly, on dynamic *logic* rather than on dynamic *probability*. In terms of dynamic probability, we could describe the *knowledge* that $12^3 = 1728$ as the ascribing of dynamic probability $p = 1$ to this mundane fact. If instead p were less than 1, then the remaining dynamic information available by calculation would be $-\log p$ (Good, 1950, p. 75; Good, 1968, p. 126). This may be compared with Michie's definition of amount of knowledge, which is based on Hartley's non-probabilistic measure of information (Hartley, 1928).

Amount of knowledge can be regarded as another quasi-utility of which weight of evidence and explicativity are examples. A measure of knowledge should be usable for putting programs in order of merit.

In a tree search, such as in chess, in theorem-proving, and in medical diagnosis, one can use entropy, or amount of information, as a quasi-utility for cutting down on the search (Good, 1970; Good and Card, 1971; Card and Good, 1974) and the test for whether this quasi-utility is sensible is whether its use agrees reasonably well with that of the principle of rationality, the maximization of expected utility. Similarly, to judge whether a measure of knowledge is a useful quasi-utility it should ultimately by compared with the type 2 principle of rationality (Appendix A). So the question arises what form this principle would take when applied to computer programs.

Suppose we have a program for evaluating a function f(x) and let's imagine for the moment that we are going to make one use of the program for calculating f(x) for some unknown value of x. Suppose that the probability that x will be the value for which we wish to evaluate the function is p(x) and let's suppose that when we wish to do this evaluation the utility of the calculation is $u(x,\lambda)$ where λ is the proportional accuracy of the result. Suppose further that the cost of obtaining this proportional accuracy for evaluating f(x), given the program, is $c(x,\lambda)$. Then the total expected utility of the program, as far as its next use is concerned, is given by the expression

$$\left.\begin{aligned} & U = \int p(x) \max_{\lambda} \{0, [u(x,\lambda) - c(x,\lambda)]\}\, dx \\ \text{or} \quad & \\ & \sum_{x} p(x) \max_{\lambda} \{0, [u(x,\lambda) - c(x,\lambda)]\}. \end{aligned}\right\} \qquad (3)$$

The notion of dynamic probability (or of rationality of type 2) is implicit in the utilities mentioned here, because, if the usual axioms of probability are assumed, the utilities would be zero because the costs of calculation are ignored. Anything calculable is "certain" in ordinary logic and so conveys no logical information, only dynamic information.

If the program is to be applied more than once, then formula (3) will apply to each of its applications unless the program is an adaptive one. By an *adaptive program* we could mean simply that *the costs of calculation tend to decrease*

when the program is used repeatedly. This will be true for example, in the adaptive rote learning programs that Donald Michie described in lectures in Blacksburg in 1974. To allow for adaptability would lead to severe complications and I suspect that similar complications would arise if Donald Michie's definition of amount of knowledge were to be applied to the same problem.

I expect his definition will usually be considerably easier to use then the expression (3), but I do not know which is the better definition on balance.

Example 1. Suppose that (i) all accurate answers have a constant utility a, and all others have zero utility. Then

$$u(x,\lambda) = \begin{cases} a & \text{if } \lambda = 0 \\ 0 & \text{otherwise;} \end{cases}$$

(ii) $c(x,0) = b$, a constant, when x belong to a set X, where $a > b > 0$, and that $c(x,0) > a$ if x does not belong to X; (iii) all values of x are equally likely a priori, that is, $p(x)$ is mathematically independent of x. Then (3) is proportional to the number of elements in X, that is, to the number of values of x that can be "profitably" computed.

Example 2.

$$u(x,\lambda) = \begin{cases} a & \text{if } \lambda < \lambda_0 \\ 0 & \text{otherwise.} \end{cases}$$

The analysis is much the same as for Example 1 and is left to the reader.

Example 3. $u(x,\lambda) = -\log \lambda (\lambda < 1)$; then the utility is approximately proportional to the number of correct significant figures.

Example 4. In the theory of numbers we would often need to modify the theory and perhaps use a utility $u(x,\mu)$, where μ is the number of decimal digits in the answer.

Example 5: knowledge measurement in chess. Let x now denote a chess position instead of a number. Let $u(x)$ denote the expected utility of the program when applied in position x, allowing this time for the costs. Then $v = \Sigma_x p(x)u(x)$ measures the expected utility of the program per move, where $p(x)$ is the probability of the occurrence of position x. The dependence between consecutive positions does not affect this formula because the expectation of a sum is always the sum of the expectations regardless of dependence. A measure of the knowledge added to the program by throwing the book on opening variations at it, can be obtained by simply subtracting the previous value of v from the new value.

It should now be clear that dynamic probability is fundamental for a theory of practical chess, and has wider applicability. Any search procedure, such as is definitely required in non-routine mathematical research, whether by humans or by machines, *must* make use of subgoals to fight the combinatorial explosion. Dynamic utilities are required in such work because, *when you set up subgoals, you should estimate their expected utility as an aid to the main goal* before you

bother your pretty head in trying to attain the subgoals.

The combinatorial explosion is often mentioned as a reason for believing in the impracticability of machine intelligence, but if this argument held water it would also show that human intelligence is impossible. Perhaps it is impossible for a human to be intelligent, but the real question is whether machines are necessarily equally unintelligent. Both human problem-solvers and pseudo-gnostical machines must use dynamic probability.

Appendix A. Philosophical applications of dynamic probability

An interesting application of dynamic probability is to a fundamental philosophical problem concerning simplicity. Many of us believe that of two scientific laws that explain the same facts, the simpler is usually the more probable. Agassi, in support of a thesis of Popper, challenged this belief by pointing out that, for example, Maxwell's equations imply Fresnel's optical laws and must therefore be not more probable, yet Maxwell's equations appear simpler. This difficulty can be succinctly resolved in terms of dynamic probability, and I believe this is the only possible way of resolving it. For the clarification of these cryptic remarks see (Good, 1968) and (Good, 1975). These papers also contain an explication and even a calculus for "explicativity," a quantitative measure of the explanatory power of a theory.

A further philosophical application of dynamic probability arises in connection with the principle of rationality, the recommendation to maximize expected utility. It frequently happens that that amount of thinking or calculation required to obey this principle completely is very great or impracticably large. Whatever its size, it is rational to allow for the costs of this effort (for example, [Good, 1971]), whatever the difficulties of laying down rules for doing so. When such allowance is made we can still try to maximize expected utility, but the probabilities, and sometimes the utilities also, are then dynamic. When a conscious attempt is made to allow for the costs we may say we are obeying the principle of rationality of type 2. This modified principle can often justify us in using the often convenient but apparently *ad hoc* and somewhat irrational methods of "non-Bayesian" statistics, that is, methods that officially disregard the use of subjective probability judgments. But such judgments are always at least implicit: all statisticians are implicit Bayesians whether they know it or not, except sometimes when they are making mistakes. (Of course Bayesians also sometimes make mistakes.)

Thus dynamic probability and dynamic utility help us to achieve a Bayes/non-Bayes synthesis. Inequality judgments rather than sharp probability judgments also contribute to this synthesis: a strict non-Bayesian should choose the interval (0,1) for all his subjective probabilities! For an interesting example of a Bayes/non-Bayes synthesis see (Good, 1967C) and (Good and Crook, 1974).

NOTES

1. Lighthill's joke, cracked in a BBC TV debate. Jokes don't wear well for long, however

risible they were originally, so I have invested a neologism that just might replace the clumsy and ambiguous "workers in A.I." The "g" of "pseudognostics" belongs to the third syllable! Michie's expression "knowledge engineering" might be preferred in some contexts, but it will tend to prevent A.I. work in any university department outside engineering. Engineering departments already tend to take the universities over.

2. Each axiom merely *relates* probability values. *Suggestions*, such as the "principle of sufficient reason," are not axioms and they require judgments about the real world.
3. By "intuitive probability" I mean either logical or subjective probability (Koopman, 1940) as contrasted with the physical probabilities that arise, for example, in quantum mechanics, or the tautological probabilities of mathematical statistics (Good, 1959A).
4. More precisely, it must be "coherent" in the sense that a "Dutch book" cannot be made against it in a gambling situation. A Dutch book is a proposed set of bets such that you will lose whatever happens (Savage, 1954).
5. Donald Michie expressed a preference for this term in conversation in 1974, since he thought that "evolving probability," which I have used in the past, was more likely to be misunderstood.
6. (i) A real mathematician, by definition, cannot do all his work by low-level routine methods; but one man's routine is another man's creativity. (ii) Two famous examples of the use of *scientific* induction in mathematics were Gauss's discoveries of the prime number theorem and of the law of quadratic reciprocity. He never succeeded in proving the first of these results.
7. Polya's writings demonstrate the truth of the aphorism in Note 6. Polya's use of probability in mathematical research is more qualitative than mine. A typical theorem in his writings is "The more confidence we placed in an incompatible rival of our conjecture, the greater will be the gain of faith in our conjecture when that rival is refuted" (Polya, 1954, vol 2, p. 124). His purely qualitative approach would prevent the application of the principle of rationality in many circumstances.
8. Presumably the ALGOL notation x: = x + 3 was introduced to avoid the apparent inconsistency.
9. It is pointless to make such judgments without some attached dynamic probabilities, so I add that I think there is a probability exceeding ½ that the machine will come in the present century. But a probability of only 1/1000 would of course justify the present expenditures.
10. Judgments are never formalized
    You can sign that with your blood gents
    For when they are formalized
    No one dare call them judgments.

    Drol Doog (With apologies to Sir John Harrington.)
11. In case this seems too obvious the reader is reminded that it was not explicit in the earlier papers on chess programming, and there is no heading "Probability" in (Sunnucks, 1970).
12. Even if every atom in the moon examined 10^{24} *games* per second (light takes about 10^{-24} sec. to traverse the diameter of an electron), it would take ten million times the age of the universe to examine 10^{100} games, which is a drop in the mare.
13. The values of the pieces also vary with the position, in anyone's book. There is much scope for conjectures and statistical work on evaluation functions. For example, it was suggested in (Good, 1967B) that the "advantage of two bishops" could be explained by assuming that it is "in general" better to control two different squares than to control one square twice, although "overprotection of the centre" might be an exception. For example, the contribution to the total "score" from the control n times of one square might be roughly proportional to $(n+1)^{\alpha} - (n+1)^{-\beta}$ $(0 < \alpha < 1, \beta > 0)$.
14. Perhaps the odds of a draw are roughly the geometric mean of those of winning and losing.

15. The International Chessmaster and senior Civil Servant, Hugh Alexander, once remarked that it is more important for a Civil Service administrator to make his mind up promptly than to reach the best decision. He might have had in mind that otherwise the administrator would "lose on the clock."
16. To be precise I said that natural language should be used, and John McCarthy said from the floor that descriptions in symbolic logic might be better.
17. This is known as psychological chess when Emanual Lasker does it, and trappy chess when I do it.
18. By definition of "intelligently."
19. The difficulty of evaluating unblocked passed pawns is one for the human as well as for the machine, because it is often in the balance whether such pawns *can* be blocked. This might be the main reason for the difficulty of formalizing endgame play. It is said that mathematicians have an advantage in the endgame but I do not know the evidence for this nor clearly why it should be true.
20. This part of the paper is based on my invited discussion of Michie's public lecture on the measurement of knowledge on October 30, 1974 in Blacksburg: see his contribution to this volume.

REFERENCES

Card, W.I. and I.J. Good (1974) A logical analysis of medicine. *A Companion to Medical Studies* (eds. R. Passmore and J.S. Robson; Oxford, Blackwell's), 3, Chapter 60.

Good, I.J. (1950) *Probability and the Weighing of Evidence* (London, Charles Griffin; New York, Hafners), p. 119.

Good, I.J. (1952) Rational decisions. *J. Roy. Statist. Soc. B*, 14, 107-114.

Good, I.J. (1959A) Kinds of probability. *Science* 129, 443-447.

Good, I.J. (1959B) Could a machine make probability judgments? *Computers and Automation* 8, 14-16 and 24-27. (Based on a lecture at the British Society for the Philosophy of Science, Jan. 1958.)

Good, I.J. (1960/68) Weight of evidence, corroboration, explanatory power, information, and the utility of experiments. *J. Roy. Statist. Soc. B*, 22, 319-331; 30, 203.

Good, I.J. (1961/62) A causal calculus *Brit. J. Philos. Sc.* 11, 305-318; 12, 43-51; 13, 88.

Good, I.J. (1967A) On the principle of total evidence. *Brit J. Philos. Sc.* 17, 319-321.

Good, I.J. (1967B) A five-year plan for automatic chess. *Machine Intelligence* 2 (eds. E. Dale and D. Michie; Oliver and Boyd), 89-118.

Good, I.J. (1967C) A Bayesian significance test for multinomial distributions. *J. Roy. Statist. Soc. B.*, 29,339-431 (with discussion).

Good, I.J. (1968) Corroboration, explanation, evolving probability, simplicity, and a sharpened razor. *Brit. J. Philos. Sc.* 19, 123-143. See also 25 (1974), 289.

Good, I.J. (1970) Some statistical methods in machine intelligence research. *Math. Biosc.* 6, 185-208. (Based on a lecture at Fort Belvoir, November 1967.)

Good, I.J. (1971) Twenty-seven principles of rationality, (Appendix to "The probabilistic explication of information, evidence, surprise, causality, explanation, and utility"). *Foundations of Statistical Inference* (eds. V.P. Godambe and D.A. Sprott; Holt, Rinehart and Winston of Canada, Ltd., Toronto and Montreal), 124-127.

Good, I.J. (1974) A little learning can be dangerous. *Brit J. Philos. Sc.* 25, 340-342.

Good, I.J. (1975) Explicativity, corroboration, and the relative odds of hypotheses. *Synthese*, 30, 39-73.

Good, I.J. and W.I. Card (1971) The diagnostic process with special reference to errors. *Methods of Information in Medicine* 10, No. 3, 176-188.

Good, I.J. and J.F. Crook (1974) The Bayes/non-Bayes compromise and the multinomial distribution. *J. Amer. Statist. Soc.* 69, 711-720.

Hartley, R.V.L. (1928) Transmission of information. *Bell System Tech. J.* 7, 535-563.

Horowitz, I.A. and G. Mott-Smith (1960) *Point Count Chess* (New York: Simon and Schuster).

Keynes, J.M. (1921) *A Treatise on Probability* (London: Macmillan).

Koopman, B.O. (1940) The basis of probability. *Bull. Amer. Math. Soc.* 46, 763-774; The axioms and algebra of intuitive probability. *Annals of Math.* 41, 269-292.

Minsky, M. and O.G. Selfridge (1961) Learning in random nets, in *Information Theory* (ed. Colin Cherry; London: Butterworths), 335-347.

von Neumann, J. and O. Morganstern (1944/47) *The Theory of Games and Economic Bahavior* (Princeton: University Press).

Pierce, C.S. (1878) The probability of induction. *Popular Science Monthly*, reprinted in *The World of Mathematics*, 2 (ed. James R. Newman; New York: Simon and Schuster, 1956), 1341-1354.

Polya, G. (1941) Heuristic reasoning and the theory of probability. *Amer. Math. Monthly*, 48, 450-465.

Polya, G. (1954) *Mathematics and Plausible Reasoning*, two vols. (Princeton: University Press).

Ramsey, F.P. (1931) *The Foundations of Mathematics* (London: Kegan Paul).

Savage, L.J. (1954) *The Foundation of Statistics* (New York and London: Wiley).

Smith, C.A.B. (1961) Consistency in statistical inference and decision, *J. Roy. Statist. Soc. B*, 23, 1-25.

Sunnucks, Anne (1970) *The Encyclopaedia of Chess* (New York: St. Martin's Press).

8

A Theory of Advice

Donald Michie

Machine Intelligence Research Unit
University of Edinburgh

Machine intelligence problems are sometimes defined as those problems which

(i) computers can't yet do, and
(ii) humans can.

In a try for a less *ad hoc* formulation we shall say that a machine intelligence problem is one whose solution program

(i) is time-infeasible if minimally represented, but
(ii) can be made time-feasible by a feasible memory extension containing "advice."

We shall further consider how much "knowledge" about a finite mathematical function can, on certain assumptions, be credited to a computer program. Although our approach is quite general, we are really only interested in programs which evaluate "semi-hard" functions, believing that the evaluation of such functions constitutes the defining aspiration of machine intelligence work. If a function is less hard than "semi-hard," then we can evaluate it by pure algorithm (trading space for time) or by pure look-up (making the opposite trade), with no need to talk of knowledge, advice, machine intelligence, or any of those things. We call such problems "standard." If however the function *is* "semi-hard," then we will be driven to construct some form of artful compromise between the two representations: without such a compromise the function will not be evaluable within practical resource limits. If the function is harder than "semi-hard," i.e. is actually "hard," then no amount of compromise can ever make feasible its evaluation by any terrestrial device.

"Hard" problems

In a recent lecture Knuth (1976) called attention to the notion of a "hard" problem as one for which solutions are computable in the theoretical sense but

not in any practical sense. For illustration he referred to the task, studied by Meyer and Stockmeyer, of determining the truth-values of statements about whole numbers expressed in a restricted logical symbolism, for example

$$\forall x\, \forall y(y \geqslant x+2 \Rightarrow \exists z(x<z \wedge z<y)), \text{ or}$$

$$\forall S(\exists x(x \epsilon S) \Rightarrow \exists y(y \epsilon S \wedge \forall z(z \epsilon S \Rightarrow y \leqslant z)))$$

Thanks to a theorem of Büchi it is known that a device could in principle be constructed capable of evaluating the truth or falsehood of any valid input expression in this symbolism in a finite number of steps. But is the problem nevertheless in some important sense "hard?"

Meyer and Stockmeyer showed that if we allow input expressions to be as long as only 617 symbols then the answer is "yes," reckoning "hardness" as follows: find an evaluation algorithm expressed as an electrical network of gates and registers such as to minimise the number of components; if this number exceeds the number of elementary particles in the observable Universe (say, 10^{125}), then the problem is "hard." A consequence follows that any representation of the same algorithm as a computer program for any sequential machine *either* would entail for some inputs too many computational steps for solution within feasible time, *or* would contain too many symbols to be accommodated in any feasible store, *or* both. In drawing my attention to this consequence my colleague David Plaisted related it to recent combinational complexity studies *(see* Schnorr, 1975, Fischer, 1975).

Our definition of "semi-hard" adopts computer programs as the sole representational form for solution algorithms, and explicitly recognises *both* categories of resource-bound, namely time (e.g. number of computational steps) and space (number of store-bits). For each given finite function we consider two different representations:

(1) *space-minimal*: the shortest program which will evaluate the function for all inputs, no concern being given to the number of computational steps required, nor to possible work-space requirements during the course of a computation.

(2) *time-minimal*: the program which requires the least number of steps for worst-case evaluation, no concern being given to the program's length or work-space.

Questions of differential weighting for different kinds of computational step are neglected. Such questions are related to a valid objection to the effect that criteria (1) and (2) cannot anyway be applied without a complete specification of the machines on which the programs are to run. In a recent outline of work in algorithmic information theory, Chaitin (1975) considers this objection in an essentially similar context and dismisses it as quantitatively unimportant. The real point here is that problems in which we are likely to be interested typically exceed our arbitrary boundary lines by such large margins that the arbitrariness hardly matters. Differences introduced by envisaging one rather than another machine specification are as irrelevant to the main thrust as would be a decision

in the earlier example to credit the Universe with 10^{120} or 10^{130} elementary particles instead of 10^{125}.

Here are some boundary lines, scaled down for coziness from Universal to planetary dimensions.

A "semi-hard" problem is not "hard"; yet any space-minimal solution-program written for any envisageable sequential machine would for some inputs require a running time greater than the age of the Earth, and any time-minimal solution-program written for any envisageable sequential machine would require a store too large to be accomodated on the Earth's surface.

But surely a problem as hard as *that* must be "hard" in the full sense of Knuth, that it "will never be solved in our lifetime, regardless of how clever people become or how many resources are committed to the project?" Not so! Here indeed is the whole joy of the phenomena of human cognition and of cognitive engineering, that a loop-hole for compromise can exist between the two criteria of minimality. How do we know, for a given "semi-hard" problem, that we cannot devise a representation which, although not minimal, is still acceptably short *and* has a non-minimal but acceptable running time? Let us do some calculations around this thought, taking our "semi-hard" function from the game of chess.

A function f links each legal chess position to its game-theoretic value; that is, it maps onto the set $\{1, 0, -1\}$ corresponding to outcomes which are "won," "drawn" or "lost" from the standpoint of the opening player. What might be a space-minimal representation of this particular f? Strictly speaking we do not know. But we can show that such a representation must be very short, for an upper bound to the minimal length is given by the iterated minimax algorithm for computing f by (i) looking ahead along all possible continuation paths from the given position, (ii) using the rules of chess to assign outcome values to all the terminal positions of the lookahead tree, and then (iii) backing these values up the game-tree using the minimax rule until the root position has been labelled. A minimum-length implementation of this as a program requires of the order of 10^4 bits only. Hence *either*

A. 10^4 bits does indeed measure the space-minimality of chess ("complexity" in Chaitin's terminology, "ω-complexity" in ours (Michie, 1976)), *or*

B. f's space-minimal representation is even shorter than this.

Case B keeps alive the possibility that chess is not even "semi-hard," since this hypothetical shortest representation might turn out to be so fast-running as to evade the time-bound horn of the dilemma. This would be the case if some sweeping mathematization of chess were discovered, analogous to the parity rule for the game of Nim.

Case A, which says that we are not going to get a shorter rule than iterated minimax, corresponds to many people's intuitions. We shall assume it here for purposes of expounding the "semi-hard" idea. Note that we are not turning aside from the possibility of a compact mathematical rule for chess, but only from the more remote possibility that such a rule might be shorter even than iterated minimax.

We first re-state our essential position.

A "semi-hard" problem is a non-"hard" problem whose space-minimal solution exceeds practical bounds of time and whose time-minimal solution exceeds practical bounds of space.

Assuming Case A, chess meets the first half of the definition. The expected waiting time for evaluating f, according to a calculation of Shannon's (1950), would exceed 10^{90} years on a machine able to calculate one variation per micro-micro-second. Let us turn then to f's time-minimal representation.

Restricting ourselves, as we have deliberately done, to computer programs for sequential machines, we adopt the look-up table as the time-minimal representation. We make any necessary assumptions concerning the relative speeds of look-up operations *versus* other operations so as to ensure that this representation comes out time-minimal in all cases. Determining the space requirement is then equivalent (disregarding the space-occupancy of the look-up program itself) to calculating the information-content of a message which encodes the extensional form of f, i.e. the sequence of symbols taken from f's co-domain Y corresponding with an ordering of the set X of size N which is f's domain. This information-content, $I(f) = -N\sum_{Y} p(y)\log_2 p(y)$, tells us the length of the shortest such message and hence the theoretical minimum number of store-bits required to hold the look-up table. The choice of ordering for the domain is treated as arbitrary, in the sense that the encoding may not exploit it to achieve additional compression, although of course free to exploit the first-order frequencies of the different y-values in the message.

On this basis the look-up table for f would require something between 10^{45} and 10^{50} bits of store, corresponding roughly to the range of the various estimates which have been made of the number of different legal chess positions. A store of this size, for any conceivable advance in micro-miniaturization, could not be assembled on the surface of the Earth. Chess, then, even in the restricted sense of recognising the game-theoretic value of a position, qualifies as at least "semi-hard."

At this point the task of evaluating f may seem a matter for despair. But all that we have so far indicated for chess is that the space-minimal solution may not be time-feasible and that the time-minimal solution is not space-feasible; *not* that no solution-program of any kind could combine the two feasibilities. Only this last case corresponds to the (unproved) hypothesis that chess is "hard."

To clarify the definition of feasibility, and to recapitulate the ideas so far discussed, Figure 1 presents "trade-off curves" for a series of hypothetical functions: two "semi-hard" functions and a "hard" function. If position-evaluation in chess is like f then error-free performance at least under correspondence chess conditions is feasible. If it is like g, then any given position can be evaluated, but at the expense in the worst case of a year's continuous running-time using a 10^{15}-bit random-access store. If it is like h, then no amount of time and memory-space within terrestrial limits would be sufficient.

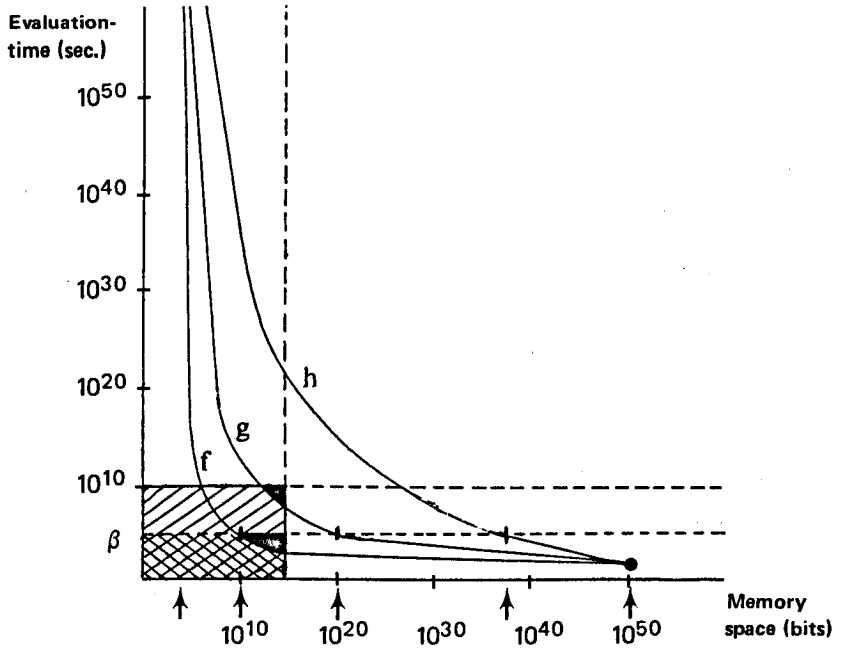


FIG. 1. Store-time trade-off curves for hypothetical finite functions f, g and h. Each has the same information-content (10^{50} bits) and the same ω-complexity (10^4 bits). Time-feasibility and space-feasibility limits have somewhat arbitrarily been placed at 10^{10} secs. and 10^{15} bits respectively. Time-acceptability is set at 10^5 secs, indicated by β. The hatched rectangle is the "zone of feasibility," through which curves f and g pass. Only f passes through the cross-hatched time-acceptability sub-zone. The five upward arrows mark respectively: the ω-complexity of f, g and h; the β-complexity of f; the β-complexity of g; the β-complexity of h; the information-content of f, g and h. f is "semi-hard," and since its β-complexity is less than the space-feasibility bound it is "not too difficult." g is "semi-hard" and "too difficult." h is "hard," which renders it *a fortiori* "too difficult."

Figure 2 depicts a special kind of "hard" function and reveals a twist in our formulation. In his paper Gregory Chaitin discusses the function which maps from major-league baseball games to their scores. "In this case" he remarks "it is most unlikely that a formula could be found for compressing the information into a short message (contrast the iterated-minimax formula for chess–D.M.); in such a series of numbers each digit is essentially an independent item of information, and it cannot be predicted from its neighbours or from some underlying rule. There is no alternative to transmitting the entire list of scores." Now suppose that there are so many major-league results (10^{50} if you like; never mind the time taken to accumulate them!) that any solution program for this function, even though time-feasible, would necessarily be space-infeasible, consisting indeed merely of a giant look-up table. The function, if we did not already know it to be "hard," would then fail to satisfy the first limb of our definition which says that a "semi-hard" problem's space-minimal solution is time-infeasible. As we progressively scale down the hardness of a problem of this

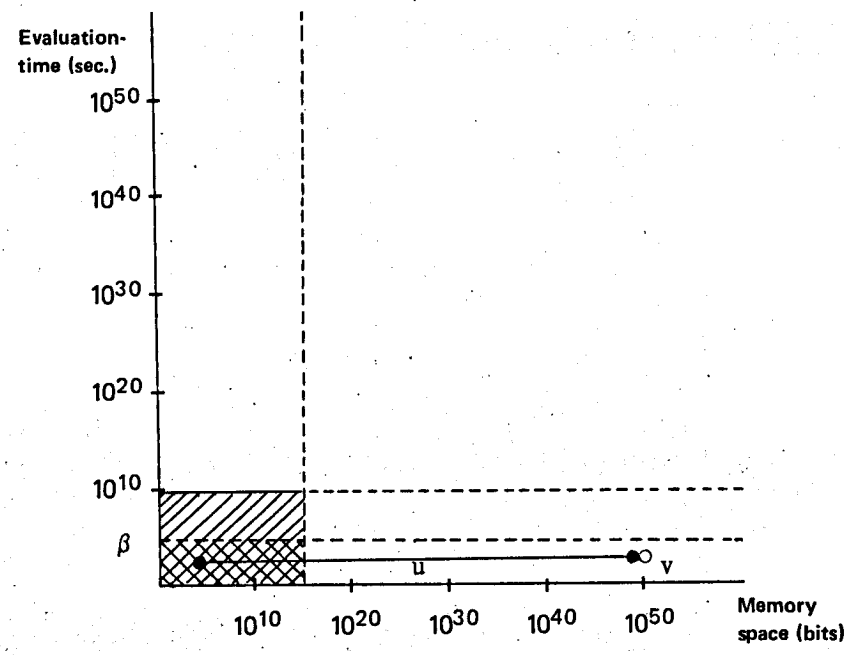


FIG. 2. Function v (open circle in the diagram) is "hard," and it cannot be compacted. Like Chaitin's "baseball" function it is devoid of internal structure and hence offers no trade-offs. Function u is of an opposite type. Its representation can be compressed into a highly compact solution program which (in contrast to iterated minimax in chess) is not appreciably slower-running than look-up. An example might be position-evaluation in games of Nim starting with, say, 17 heaps of 1000 pieces each.

type by gradually decreasing the domain size, the transition from "hard" to "standard" occurs abruptly without passing through a zone of "semi-hardness." Possessing no exploitable structure, such problems offer no hand-holds to machine (or any other) intelligence. The Figure also shows, for contrast, a "standard" problem of the super-exploitable "Nim" type.

Our definition may not seem entirely satisfactory, since it catches in its net problems which we have come to regard informally as not so difficult, for example sorting a list. It may well be that the space-minimal solution of the sorting problem is none other than the simpleton program which repeatedly traverses the whole list, swapping neighbour pairs whenever the sort relation shows them to be the wrong way round—time-infeasible for quite a modest list size. Hence even though numerous fast methods are known today which would make light work of it, sorting such a list shows up as "semi-hard" on our definition. So be it. We often forget how difficult a problem really is once it has succumbed to an intensive search for good solution methods (see also the recent appearance of sorting as a domain for machine intelligence work, as in Barstow and Green, this volume).

We now turn to "advice," seen as an approach to the solution of semi-hard problems.

Advice

Suppose that we have a space-minimal program, or indeed any short naive program which is time-infeasible, and we want to make it feasible. Two contrasted paths offer themselves, the path of the Great Leap Forward and the path of Incremental Advice.

Great Leap Forward. We scrap our naive program. Then after profound analysis of the problem we write a somewhat longer program expressing a fundamentally new algorithm. Here are three examples.

1. Sorting large lists. A naive program sorts by next-neighbour swapping. Hoare then creates Quicksort.

2. Factorizing large integers (like $2^{128} + 1$). A naive program counts up the number series doing divisions, first into the input number, then into the divisor and quotient found by successful division and so on. Solution times for a superfast machine would be measured in thousands of years. Brillhart and Morrison (cited by Knuth) devise a program to do it in an hour or two "by a combination of sophisticated methods, representing a culmination of mathematical developments which began about 160 years earlier."

3. Chess. A naive program executes the iterated-minimax rule, with a running time in the region of 10^{90} years. Some genius yet unborn devises a formula for identifying positions according to their game-theoretic value. Let us suppose that the new formula, unlike the Nim rule, is bulkier to represent than iterated minimax, but that it is nonetheless reasonably compact, and time-feasible.

The Great Leap Forward is without doubt the path of honour. Possibly progress towards solution of most "semi-hard" problems can and should be made in this way. Sometimes, however, we cannot wait for someone to make the Leap; or quite simply we are impressed by the fact that human expert performers in the given domain compute quite good solutions (as in chess) by methods which give no evidence at all of the kind of unitary insight associated with a Great Leap Forward. Also, there is the risk that we might have to wait forever, since some of the intellectual skills of man may owe their mosaic quality not to the limitations of their possessors but to lack of any deep structure in the given problem-domain for mathematical insight to seize upon.

Incremental Advice. We retain our naive program essentially unaltered, but from time to time we add to the store new materials which do not of themselves perform any computations necessary to evaluation but which act solely to expedite the operations of the naive evaluation program. This catalytic material is "advice." To illustrate we now consider the prime-counting function, which, given an integer, returns the number of primes less than that integer. The evaluation of this function is not the kind of problem normally associated with machine intelligence work. But Watterberg and Segre have analysed this straightforward and classical numerical problem in such a way as to display the Incremental Advice idea in a peculiarly simple and compelling fashion. They also nicely demonstrated a special advantage of the "advice" approach, namely that it directly lends itself not only to incremental additions by the user but also by

the program itself ("learning"). The following account is excerpted from their own report of the work (Watterberg and Segre, 1976).

> The work was done using a Digital Equipment Corporation PDP-11/35 running the UNIX timesharing operating system. All the programs were written in either PDP-11/35 assembly language or the UNIX system language, "C." Due to the nature of the UNIX timesharing system, the machine specification, S, on which the test program were executed, is simply a PDP-11/35 with 26K bytes of core memory. For information on instruction times and core cycle times, see the PDP-11/35 reference manual and the PDP-11/35 peripherals handbook.
>
> Three different function evaluation programs were written. Each had a different level of advice and/or learning ability. A short description of each follows.
>
> Program 1. Pure algorithmic
>
> In order to evaluate the function this program considers each integer from 2 to the function argument (P) minus 1 as a prime-candidate. Each candidate is tested for "primeness" by a division algorithm with the integers from 2 to sqrt(P) as the divisors. The prime-testing function is iterative and has no memory of any previous calls to it.
>
> Program 2. Advice
>
> Identical to Program 1 with advice added. The advice provided is an extension of the obvious fact that (apart from 2) even integers are not prime. If the multiples of 3 and the multiples of 5 are also removed from consideration, 8 integers in 30 remain as prime candidates as follows:
>
> Any integer can be expressed as
>
> $$30n + k, \quad 0 \leqslant k < 30.$$
>
> Since 30 is divisible by 2, 3, and 5, the only integers that might be prime are those for which k is not divisible by 2, 3, or 5, i.e. k = 1, 7, 11, 13, 17, 19, 23, 29.
>
> This advice was implemented as a gap table (consisting of the 8 gaps) which kept track of how much to add to the last prime-candidate to obtain the next prime-candidate. Note that this advice is also of use in the prime-testing routine since only those 8 in 30 numbers need be used as divisors.
>
> Program 3. Advice and Rote Learning
>
> Identical to Program 2 with a rote dictionary added. The rote dictionary contains ordered pairs of integers remembered from previous calls to the program. The rote dictionary is consulted before computation begins to find the largest ordered pair less than P. This value of the co-domain is used as a starting point for further compu-

	β	Program 1	Program 2	Program 3 (25 trials)	Program 3 (250 trials)
X	10 secs.	2050	6 560	74,213	399,487
	20 secs.	3630	11,160	125,201	676,589
	30 secs.	5100	15,200	166,159	903,589
L		5.15×10^3	7.0×10^3	12.1×10^3	26.5×10^3

TABLE 1. Data from Watterberg and Segre's programs 1, 2, and 3. X is the largest x for which the program could evaluate the function within β seconds. L is the store-occupancy of the given program. The last two columns relate to "learning" periods of 25 and 250 trials respectively. These experiments were run with a helpful tutor, i.e. the questions were put to the program in an order which maximised its learning rate.

> tation. A more efficient scheme would have required one half the rote storage by finding the closest ordered pair and counting up or down to the desired value.

Table 1 shows a sample of their tabulated results for programs, 1, 2 and 3 using three different time cut-offs for the experiment, namely 10, 20 and 30 seconds, i.e. for the purpose of this scaled-down laboratory study the criterion "time-feasible" was replaced by time cut-offs fixed at these levels. The first two columns of the Table make plain that a relatively small increment of advice to the store (raising the core-occupancy from 5,150 to 7,000 bits) yields a relatively large gain in the performance of the system, roughly tripling the range over which the prime-counter can be evaluated within the time-limit. The last two columns show the even more dramatic gains obtained by adding to the fixed advice a program-incrementable dynamic dictionary of rote-knowledge.

Rote-knowledge is of course the lowliest and least interesting of all forms of knowledge. Watterberg and Segre go on to speculate whether, retaining the "naive program *versus* advice" dichotomy, higher-level concepts could in principle be incorporated in the advice, even to the level of Hardy and Wright's (1938) "prime number theorem": $\lim_{x \to \infty} \frac{f(x)}{x/\ln x} = 1$, where $f(x)$ = number of primes less than x.

Formalising "knowledge"

This same dichotomy between naive program and advice has been made the basis of a package, AL1, for generating chess end-game strategies from Tables. Preliminary accounts are available elsewhere (e.g. Michie, 1976a). The present concern is that this style of conveying human knowledge to programs demands more precise and quantitative ways than are at present available for talking

about the knowledge-content of programs and adjoined advice. As we build and unbuild the advice, adding, modifying, or deleting rote-entries, descriptions, heuristics, theorems and the rest, we would like to be able to say of each separate item just what benefit it confers in terms of added knowledge and what is its cost in terms of added store-occupancy. A method of cost-benefit analysis will be sketched. But first it is desirable to have a clean-cut basis for definition and measurement of "knowledge." When this ground has been cleared, the philosophy can be forgotten.

Philosophers of the knowledge problem have always agreed that at least those facts which are explicitly and retrievably stored in memory are "known." What of facts which are stored implicitly, retrievable only by deduction? The earliest treatment of the problem seems to be the opening passage of Aristotle's *Analytica Posteriora*. He recognizes three levels:

universal knowledge of the *class* of facts implicit in the stored premisses (e.g. that all triangles have angles equal to two right angles);

virtual knowledge of every particular such fact which is deducible from the universal but has not yet either been deduced or acquired directly from sense-impression (e.g. that some particular triangle, which we have not yet encountered, has angles equal to two right angles);

unqualified knowledge of a particular fact which *has* been so acquired (e.g. that some particular triangle, which we have encountered, recognized, and thought about, has angles equal to two right angles).

In respect of his "virtual" category, Aristotle brings up some particular cases of difficulty, including the following. Consider a student's "knowledge" of the results of evaluating the predicate "having angles equal to two right angles" for an individual member of the class "triangle." The evaluation is conceived as proceeding in three stages:

(1) the instance is presented;
(2) it is recognized as belonging to the class "triangle;"
(3) the conclusion is drawn that the predicate is true of this particular instance.

Aristotle comments: "For example, the student knew beforehand that the angles of every triangle are equal to two right angles; but it was only at the actual moment at which he was being led on to recognize this as true in the instance before him that he came to know 'this figure inscribed in the semi-circle' to be a triangle... Before he was led on to recognition or before he actually drew a conclusion, we should perhaps say that in a manner he knew, in a manner not.

"If he did not in an unqualified sense of the term *know* the existence of this triangle, how could he know without qualification that its angles were equal to two right angles? No: clearly he *knows* not without qualification but only in the sense that he *knows* universally..." For consistency with context we must read the last sentence as though it continued, "... that this is true of the class

	Case A	Case B	Case C
Question asked:	Is 19 prime?	Is 199 prime?	Is 19,999,999 prime?
Stage of transition from virtual to unqualified knowledge en route to stage (5)	(3)	(4)	In practice never

TABLE 2. Three cases of the attempt to retrieve facts about the primeness of numbers. The numbers in parentheses refer to labelled stages in the text.

of triangles, and hence *knows* virtually that it is true of this triangle."

Aristotle's notion of a state of knowledge as capable of evolving dynamically under the sole influence of internal operations is a suitable point of departure for computational approaches, such as the present theory or I.J. Good's "dynamic probability" (this volume). But Aristotle's definitional system is a little too restrictive and we shall now test it to destruction by dropping increasingly heavy weights upon his "virtual" category. Then we re-build it so as to explicate knowledge in an unrestricted computational framework.

Aristotle remarked on the virtual category as being in some sense ambiguous and seemed unhappy with it. Any sense of dissatisfaction or unclearness (" . . . we should perhaps say that in a manner he knew, in a manner not") was, we propose, rooted in his unfamiliarity with computational ideas, in particular the notion that some facts may require impracticably lengthy calculations for their demonstration.

Turning from triangles to integers, consider again the predicate "prime." We shall suppose that the student evaluates it for a given integer n by testing for divisibility by unit increments of i where $1 < i^2 \leqslant n$. We shall also suppose that the student possesses rote knowledge of a few primes. Now consider the following stages:

(1) he noted that a candidate instance has been presented;
(2) he recognizes it as belonging to the class "integer;"
(3) he matches it against rote memory and if successful he goes to (5);
(4) he attempts to deduce (by evaluating "prime") that the predicate is true of this particular instance;
(5) he gives the answer.

For three superficially similar questions the student's responses might be as shown in Table 2.

Unless our student is a rare calculating prodigy we shall in case C eventually weary of waiting for his answer. So how can we ever credit him with "unqualified" knowledge that 19,999,999 is prime? Not until he has completed his calculation, which in practical terms he never does.

Aristotle's system of inference involved only computationally light syllogistic forms. If he had clearly envisaged the possibility that some straightforward questions might so draw out the calculative chains as to preclude an answer within the questioner's lifetime he would surely not have allowed the "primes" student the same knowledge status as the "triangles" student. Neither shall we. Instead we sub-divide the facts which a man "virtually" knows into (case B) those which are both virtually and practically known and (case C) those which are virtually but *not* practically known. In our system the former will be accepted as "known" but the latter are definitely *not known*. Such facts might subsequently achieve "known" status either through a modification of the knowledge-base which sufficiently shortens their derivation chains (e.g. addition of a key lemma), or by modification of the system of inference itself in the direction of improved efficiency so that some previously impracticable derivations become practicable, or by independent acquisition from an external source.

The "known" is thus equated to what can (practically) be retrieved, and Aristotle's "in a manner he knew, in a manner not" is replaced by "if able to answer with acceptable speed he knew, otherwise not."

Now we tie this down to specifics, and propose a workable calculus for measuring the knowledge-content of computer programs and associated bodies of advice.

Numerical measurement of knowledge

We interpret knowledge as the ability to answer questions, and we equate question-answering to the evaluation of finite functions. All questions in our system are expressible in the form "What is the value of f(x)?" All "answers" take the form "y", where y is an element of f's co-domain. We diverge from Aristotle in supposing that for some x's in X the given evaluation device (whose "knowledge" of f we are concerned to measure) may deliver the corresponding y-value in acceptable time but for other x's not. This time-bound, which is set by the questioner, is denoted by the symbol β.

β need not be of the cosmological magnitudes which we earlier used to dramatise the notion of "time-infeasibility": for example in the context of speed chess the user might set β to a few seconds.

It now becomes meaningful, having specified a machine, a program, and a particular value of β (expressed either in time-units, or in number of computational steps) to speak about some given f in the following fashion:

f is β-evaluable for x_1;
f is β-evaluable for x_2;
f is *not* β-evaluable for x_3;
f is β-evaluable for x_4;
f is *not* β-evaluable for x_5;
.
.
.
etc.

Consider now the partial function $f_K: X \rightarrow Y$, which is defined only for arguments in that sub-set of X, X_K, for which f is β-evaluable, i.e. in the above example $\{x_1, x_2, x_4, \ldots\}$. f_K corresponds to the *known part* of f. We can write it, in the terms of the above example, as $((x_1,y_1),(x_2,y_2),(x_4,y_4),\ldots)$. Taking the ordering of X as given, the string $y_1y_2y_4\ldots$ conveys the identical information. To determine the information-content of this string viewed as a classical information-theoretic message, we regard the constituent symbols as having been sampled from an alphabet consisting of the set Y with probabilities given by the relative frequencies with which the various symbols appear as right-hand elements of f's function table: $((x_1,y_1),(x_2,y_2),\ldots(x_N,y_N))$. [Remember that for a many-one function, size (Y)<size(X) and hence $y_i = y_j$ for some (i,j)'s. For example, f might be a Boolean function.] Earlier we wrote

$$I(f) = -N \sum_{Y} p(y)\log_2 p(y) \qquad (1)$$

for the information-content of f, where the expression p(y) is to be evaluated exactly as we have just described. But for the information-content of a designated fragment of f, such as its "known part" f_K, this expression is not suitable. The trouble is that $-\sum_{Y} p(y)\log_2 p(y)$ is an *average*, namely the average information-content per symbol. This average is then multiplied by N to obtain the entire message's information-content. If we want to deal in fragments we need a formula which associates with *each constituent symbol* its own proper information-content. The information-content of any sub-message whatsoever can then be found simply by summing directly over its constituent symbols.

The information-content of an individual symbol has been termed by Samson (1951) its "surprisal." For the message's r^{th} symbol it is $-\log_2 p(y_r)$. We accordingly re-write I(f) as a sum of N surprisals, thus:

$$I(f) = -\sum_{i=1}^{N} \log_2 p(y_i) \qquad (2)$$

It now follows naturally that the information-content associated with f's known part, f_K, should be

$$I(f_K) = -\sum_{i=1}^{N_K} \log_2 p(y_i) + N^* \qquad (3)$$

Where the successive values i = 1, 2, 3, ... index the right-hand elements of the 1st, 2nd, 3rd, ... members of f_K (not of f). $p(y_i)$ is reckoned as before from the frequency of the symbol y_i in f's function table (not f_K's). We thus arrive at a definition of the amount of knowledge about f as equal to *the information-content associated with f's known part*, i.e.

$$K(f) = I(f_K) \qquad (4)$$

This identification can be criticised on the grounds that the simple summa-

*N is the number of bits required to specify the binary condition "known/unknown" for each of the N members of f's domain.

tion of surprisals attaches equal weight to each, whereas the answers to some questions may be more useful to the questioner than the answers to others. From this standpoint a program for evaluating the chess function should surely not receive credit for its knowledge of the game-theoretic values of the "obvious" cases, which occupy most of the state space, on a scale equal to the knowledge which it displays when questioned on "interesting" positions of the kind which might arise in actual play. We meet this objection by refining (4) above, replacing it with

$$K(f) = -\sum_{i=1}^{N_K} u_i \log_2 p(y_i) \tag{5}$$

where the u_i's are utilities, normalised to have unit mean.

Finally, we relate benefit, in the form of useful knowledge as just defined, to the bit-cost of storing the program, L(f), and obtain the program's "computational advantage"·

$$C(f) = \frac{K(f)}{L(f)} \tag{6}$$

Note that for a time-minimal representation as a minimally encoded look-up table, C(f) is equal to unity, (since if $f_K = f$ then $K(f) = I(f) = L^*(f)$, where L^* is the bit-cost of the minimally (non-redundantly) represented table).... In what follows we drop the constant argument f and for convenience write K, L, . . . etc.

Knowledge-content of advice: a worked example

Our purpose in setting up this formalism is to be able to say, when a given body of advice p_B is added to the store and enabled to communicate with a naive solution program p_A, whether the advice has done some good, and how much good. To determine this we measure the cost-benefit parameters K and L both for the augmented program p_A plus p_B, and for p_A alone. The differences give us K_B, the amount of p's knowledge about f relative to p_A, and L_B, the bit-cost of p_B. K_B/L_B is then a cost-benefit ratio associated with the given body of advice. We refer to it as the system's "advisory advantage."

Actually it is a little more complicated than has just been suggested, because we must also take account of the cost-benefit parameters of an unavoidable "extra," namely the control program p_C needed to mediate communication between p_A and p_B. Strictly, the measurements of K and L for the unadvised program must be made with p_C loaded, and it is the measurements on p_A plus p_C which are to be subtracted to arrive at the final quantities for p_B. The relevant relations are

$$K = K_A + K_B + K_C \text{ (overall knowledge)} \tag{7}$$

$$L = L_A + L_B + L_C \text{ (overall store cost)} \tag{8}$$

$$C = \frac{K}{L} = \frac{K_A + K_B + K_C}{L_A + L_B + L_C} \text{ (overall computational advantage)} \tag{9}$$

	β	Program 1 A	Program 1 B	Program 1 C	Program 2 A+B+C	Program 3 (25 trials)	Program 3 (250 trials)
K	10 secs.	17×10^3	45.3×10^3	$-.3 \times 10^3$	62×10^3	$.95 \times 10^6$	6.0×10^6
K	20 secs.	33×10^3	85.5×10^3	$-.5 \times 10^3$	118×10^3	1.74×10^6	10.7×10^6
K	30 secs.	47×10^3	116.0×10^3	-1.0×10^3	162×10^3	2.38×10^6	14.6×10^6
L		5.15×10^3	$.25 \times 10^3$	1.6×10^3	7.0×10^3	12.1×10^3	26.5×10^3
K/L	10 secs.	3.3	181	–	8.8	78	226
K/L	20 secs.	6.4	330	–	16.9	144	402
K/L	30 secs.	9.2	464	–	23.1	196	549

TABLE 3. Knowledge, bit-costs and computational advantage measurements for Watterberg and Segre's programs 1, 2, and 3. Columns B and C were obtained by running program 2 with no advice, subtracting from column A+B+C to obtain column B and subtracting column A to obtain column C (see text).

K_C is always negative, and can be thought of as the "knowledge-overhead" associated with p_C. It is measured by running p_A with and without p_C and taking the difference between the amounts of knowledge measurable in the two cases.

Returning to the experiment of Watterberg and Segre we reproduce in Table 3 the values of these quantities as measured for the "prime counter" program with and without various increments of advice.

The tabulation brings out very clearly the feasibility and attractions of partitioning knowledge and costs parameters among different subdivisions of store, according to the "Incremental Advice" approach discussed earlier. In the last columns of the Table the impact can be noted of enabling the program to increment its own advice as part of an elementary rote-learning scheme. A finer partitioning of advisory knowledge and cost (not shown here) into components specifically associated with the rote-dictionary as distinct from the fixed body of advice reveals that the transition from 25 to 250 learning trials, although conferring almost a threefold further increase in overall computational advantage is approaching a point of diminishing returns: the computational advantage associated strictly with the rote-dictionary starts to fall. For details the reader is referred to the original paper.

As an aid to gaining an intuitive grasp, we should briefly consider the meaning of the K/L ratio when it characterises a body of advice, or an increment added to an existing body of advice. Clearly the ratio relates the amount of additional knowledge to the additional store-cost. But there is another way of looking at it which may possibly be illuminating. K_B/L_B is actually a ratio of compression. It tells us how much *less* store is consumed by adding the given knowledge to the system in the form of the given advice than would be consumed if the same knowledge were added in the form of a minimally (non-redundantly) coded lookup table for the partial function f_B. f_B of course consists of just those additional (x,y) pairs which become "known" when p_B is added to the system, these not having been "known" before .

Measuring a problem's "difficulty"

The earlier discussion of "semi-hard" problems did not extend to the quantitative measurement of degrees of hardness for such problems. The formalism which we have sketched now places us in a good position to do this. For a given f we consider the length of the shortest program possessing complete knowledge of the function: *not*, be it noted, the shortest program possessing complete information about the function; this would be equivalent to the function's Chaitin-Solomonoff-Kolmogorov "complexity" (*see* Chaitin, 1975), and since we have seen that for chess this quantity comes out to a mere 10^4 bits it plainly will not do as an index of hardness. Instead we take the function's *β-complexity*, i.e. the minimum bit-cost of a program possessing complete knowledge of f, I(f) in amount, and we call it f's "difficulty." If it comes to a very large number, corresponding to an infeasible store size, we say that f is "too difficult" for the

given β. Note that a "semi-hard" problem may be "too difficult" or it may be "not too difficult." Observations on the performance of chessmasters, and calculations as to the largest L_B that could be input to the human brain in a lifetime within the known limits to rates of information-transfer within the nervous system, encourage the hope that chess may turn out after all to be "not too difficult."

Knuth discusses the strategy when faced with a hard problem of accepting an evaluation mechanism with a bounded level of error. Extensions of the formalism presented here deal with certain forms of erroneous evaluation, and also with the notion that some degree of knowledge can be attached to a computation which is truncated by the β cut-off before completion and yet has succeeded during that time in shrinking the set of candidate answers. These and other details and elaborations will be found in the full account of the theory (Michie, 1976).

Note on "semi-hard" problems

It will reasonably be objected that the definition of "semi-hard" is needlessly disabled from corresponding with most people's idea of a machine intelligence problem, as a result of its critical dependence on concepts of "space-minimality" and "time-minimality." No one in the real world tries to make his program the shortest possible, nor would a sane man cling to time-minimality at the expense of a huge look-up table if by a small relaxation in running times he could obtain major savings of store. Problems may therefore exist which are easily soluble by conventional programming approaches, yet which can be made to look horrendous by rigid application of our definitions.

The remedy is to allow whoever wishes to use these definitions the freedom to blur their edges. Let him substitute "almost minimal" for "minimal" where he pleases, together with whatever tolerance in approximating the minimal (within a factor of 2, within a factor of 10, etc.) seems to him appropriate. Obviously the spirit of "semi-hard" is not met (nor does the need arise in such a case to assemble in store large bodies of advice) by a function whose trade-off curve is such as to permit reasonably fast evaluation by a near-minimal program, but for which an abrupt transition to time-infeasibility occurs as soon as the program is required to be actually minimal. We cannot ignore the possible prevalence and practical importance of such functions. Strictly, then, a definition which aims to be useful should filter them out.

Future work

The game of chess offers a domain which is finite, formally defined, at least "semi-hard," possibly "hard," and (most important) readily decomposable into sub-domains which can be isolated for separate study and measurement. Evaluation mechanisms exist in the brains of Grandmasters. Although not infallible, these can evaluate chess positions over most of the domain with an impressively low level of error. Studies by cognitive psychologists have shown these mech-

anisms to be "advice-driven" rather than "search-driven" and a massive literature has accumulated over centuries in which the masters have attempted to describe this advice. A challenge exists here to the machine intelligence specialist to translate and elaborate chess advice into machine representations which are precise and complete by the test of correct play against masters. A pure look-up (and correspondingly bulky) program for King, Rook and Pawn *versus* King and Rook written by Arlazaroff was recently validated in such a test, adjudicated by Grandmaster Awerbach (see *Firbush News 6*, (ed. J.E. Michie), Univ. of Edinburgh). Store-occupancy was of the order of a thousand megabytes. Systematic application to this domain of the framework which we have developed might be rewarding, proceeding backwards from the end of the game through the "foothills" as it were, i.e. via KRK, KPK, KQK, KQKR, KPKR, etc., these being specimen sub-domains into which KRPKR decomposes by loss or promotion. The thrust should be towards numerically characterising information-content and difficulty-bounds of individual sub-games, and measuring parameters relating to knowledge, cost and store-compression for each incremental component of advice. A number of examples and discussions of "foothill studies" have been brought together by M.R. Clarke in a recent book (*Advances in Computer Chess 1* (ed. M.R. Clarke), Edinburgh University Press).

REFERENCES

Chaitin, G.J. (1975) Randomness and mathematical proof. *Scientific American*, **232**, 47-52 (May, 1975).

Clarke, M.R. (ed.) *Advances in Computer Chess 1*, Edinburgh: Edinburgh University Press, 1977.

Fischer, M.J., reviews relevant network complexity results in *MAC Memorandum No. 65*, M.I.T., 1975.

Hardy, G.H. and Wright, E.M. (1938) *An Introduction to the Theory of Numbers*. Oxford University Press.

Knuth, D.E. (1976) Mathematics and computer science: coping with finiteness. *STAN-CS-76-541*. Stanford University.

Michie, D. (1976) Measuring the knowledge-content of programs. *Technical Report UIUCDCS-R-76-786*. University of Illinois, Computer Science Department.

Michie, D. (1976a) AL1: a package for generating strategies from tables. *SIGART Newsletter No. 59*, 12-14.

Samson, E.W. (1951) Fundamental natural concepts of information theory. *Report E5079*, Air Force Cambridge Research Station.

Schnorr, C.B. (1975) The network complexity of equivalence and other applications of network complexity. *Fachbereich Mathematik*, Universität Frankfurt.

Shannon, C.E. (1950) Programming a computer for playing chess. *Phil. Mag.* **41**, 256-275.

Watterberg, P. and Segre, C. (1976) Knowledge measurement. *CS397DM Group Report*, University of Illinois, Computer Science Department.

INDUCTIVE ACQUISITION OF KNOWLEDGE

9

Rationality, Evidence, and Induction in Scientific Inference

I. J. Good

Virginia Polytechnic Institute and State University
Blacksburg, Virginia

[Opening remarks from the Chair at a specially convened discussion on inductive inference: Dr. Michalski's paper in this volume was a prepared contribution to this same discussion—Editors.]

The topic is closely related to my earlier paper (Good, 1968B). In that paper "duality" referred to the process of working both up and down trees, a topic that has been mentioned a few times already in the present conference. It might have been logical to repeat my paper here, but there are other topics, relevant to this conference, that have interested me such as (i) Explicativity or the Sharpened Razor, (ii) Hypothesis formulation; (iii) Rationality; (iv) Evidence; (v) Complexity; (vi) Scientific Induction; (vii) Kinds of Probability; and (viii) Probabilistic Causality. For my scribblings on these topics I refer you, for example, to (Good, 1950, 1952, 1959, 1961, 1963, 1968A, 1974, 1975A, 1975B).

During the Panel Discussion at the end of the conference, Peter Suzman asked whether we should be concerned with Knowledge Systems or with Belief Systems. My reply was that the latter is more complete, and this view is implicit in my paper on dynamic probability. Nonprobabilistic logic is the skeleton of machine intelligence, and probability is its flesh. This does not imply that probability is more important than formal logic, only that it is more ultimate. As I see them, all the topics (i) to (viii) are based on probability.

Topics (i), (ii), (v), and (vi) are especially close to that of today's colloquium; topic (v) to Sharon Sickel's paper; (iii) and (iv) to the measurement of knowledge and to MYCIN; topic (vii) helps to clarify all the other seven topics; and topic (viii) is close to that of the paper by Schank and of (Berliner, 1974).

Instead of trying to paraphrase (Good, 1950, 1952, 1959, 1961, 1963, 1968A, 1974, 1975A, 1975B), in this printed version of my presentation, it seems more appropriate to make some points that are not in those publications. (In the spoken version I concentrated largely on the notion of explicativity [Good, 1975A]).

In (Good, 1974) I said that the complexity of a proposition could not be defined as - log p, where p is its prior probability, because, for one thing, this would make the proposition 0 = 1 have infinite complexity. A proposition and its negation should have approximately equal measures of complexity, and the complexity of the conjunction of two entirely independent propositions should be roughly equal to the sum of their complexities. Accordingly I suggested that the formula - log p might be reasonable if p is interpreted as the probability, *as a linguistic text,* of the most economical linguistic expression of the proposition, where, by the way, the language should be optimally economical for the general context. But in the discussion, Peter Suzman asked whether "Caterpillars have chromosomes" should be regarded as more complex than "Dogs have chromosomes" and I think the answer is "No." So the above definition should be regarded as only a first approximation. Nor is it good enough merely to count parameters; because, for example, an inverse 2.13th power law is more complicated than an inverse square law. We haven't arrived yet at an entirely satisfactory definition of complexity, but brevity certainly has something to do with it as, for example, Lemoine (Lemoine, 1902) emphasized in a geometrical context.

In my opinion the main reason we should be interested in measuring complexity, when formulating hypotheses, is only because complexity has *some* relationship to prior probability. If we could entirely trust our judgment of the prior probability we could throw away the ladder of complexity that may have helped us to arrive at that judgment.

Another matter I'd like to put on record here is a simple application of the principle of rationality to the funding of research on machine intelligence. Presumably many of you have seen a recording of the television debate on this topic that was put out by the British Broadcasting Corporation. That debate provoked me to write a letter in October 1973 to Sir James Lighthill of which I here quote the complete text:

> "Dear Sir James,
>
> "It is now more than six years since we had those meetings* about the possibility of a school for highly gifted children in Britain. Can you tell me what became of our suggestions?
>
> "I happened to be in the United Kingdom last month and I heard your T.V. debate with Donald Michie and Co. and I enjoyed the lively and humorous discussion. When you stated that in your opinion artificial intelligence at the level of a human is a mirage, at least for say the next sixty or seventy years, would it be possible for you to state some degree of confidence that you have in this judgment? I don't know whether you are prepared to estimate a range of values for your subjective probability in the matter. As I believe you

*These meetings were organized by Lighthill and Bryan Thwaites, and were attended by M. Young, A.C. Offord, and myself.

know, I would put the probability high, for example, at least 0.75.

"One of your arguments was that much less progress has been made in the last twenty years than was expected by some of us twenty years ago. I do not regard this as a strong argument because all that anyone could have meant by predictions made at that time was some kind of estimate of the probability distribution of the time taken to reach the "ultraintelligent machine." If the probability of its being achieved is estimated as a ½ and it is then not achieved this counts as only a factor of at most 2 against the original prediction as compared with the prediciton that it would certainly not be achieved. A factor of 2 is small beer.

"That the task is a difficult one can be seen in a number of ways. One of them is that it took fifteen man-years to write Fortran. . . . programming is much more complicated than people thought it would be in 1947. I suppose that very few people in 1947 would have predicted that by now there would be at least half a million computer programmers in the world. (In fact the opposite view was expressed by some people, namely that the computer would put numerical analysts out of work!)"

"With kind regards,
"Yours sincerely, etc."

In his reply Sir James said that in a television programme it is impossible to discuss matters in the depth that is desirable and he referred me to (Science Research Council, 1973) where his views are more completely given. But even there he does not state a ball-park estimate of the probability I requested. Let us then suppose that the word "mirage" implies that the probability is less than 1/100. Now the value of a machine with human intelligence, since it would quickly lead to an ultraintelligent machine, is obviously worth as much, in absolute value, as the gross international product, a few trillion dollars, though it may be negative. Therefore, if we accept my estimate of Lighthill's implicit pessimistic upper estimate of the probability, it follows that the investment should not exceed some amount in excess of some tens of billions of dollars. This does not support a case for cutting the present expenditures! The real case against the investment must be, not that the UIM is a mirage, but that its value might be negative. But the *absence* of a UIM looks likely to lead to the destruction of civilization anyway, so if anyone is prepared to give me a billion dollars I'm prepared to start a UIM foundation.

REFERENCES

Berliner, Hans (1974) Chess as problem solving: the development of a tactics analyzer. *Ph.D. Thesis,* Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Coolidge, J.L. (1961). *A Treatise on the Circle and the Sphere.*

Good, I.J. (1950) *Probability and the Weighing of Evidence* (London, Charles Griffin; New York, Hafners).

Good, I.J. (1952) Rational decisions, *J. Royal Stat. Soc B* **14**, 107-114.
Good, I.J. (1959) Kinds of probability. *Science* **129**, 443-447.
Good, I.J. (1961) A causal calculus, *B. J. Phil. Sci.* **11**, 305-318; **12**, 43-51; **13** (1962), 88.
Good, I.J. (1963) Maximum entropy for hypothesis formulation, especially for multi-dimensional contingency tables, *Ann. Math. Statist.* **34**, 911-934.
Good, I.J. (1968A) Corroboration, explanation, evolving probability, simplicity, and a sharpened razor, *B. J. Phil. Sci.* **19** 156-157.
Good, I.J. (1968B) Creativity and duality in perception and recall *I.E.E./N.P.L. Conference on Pattern Recognition, July 1968* (Institution of Electrical Engineers, London), 228-237.
Good, I.J. (1974) A correction concerning complexity, *B. J. Phil. Sci.* **25**, 3, 289.
Good, I.J. (1975A). Explicativity, corroboration, and the relative odds of hypotheses *Synthese*, **30**, 39-73.
Good, I.J. (1975B). The Bayesian influence, or how to sweep subjectivism under the carpet, in a 1973 conference to be published in *Foundations and Philosophy of Statistical Theories in the Physical Sciences* (ed. W.L. Harper and C.A. Hooker; Dordrecht: D. Reidel).
Lemoine, E. (1902) *Géométrographie*, cited in (Coolidge, 1961).
Science Research Council (U.K.). (1973). *Artificial Intelligence: a Paper Symposium.*

10

An Experiment on Inductive Learning in Chess End Games

R. S. Michalski and Pericles Negri

Department of Computer Science
University of Illinois at Urbana-Champaign

INTRODUCTION

Further progress in the application of computers to many practical fields seems to depend heavily on the success in implementing learning and inductive processes within machines. For example, to develop a consultation system for medical or plant disease diagnosis, prognosis and decision making in general, it is very desirable, perhaps even necessary, to be able to 'teach' the system through examples of correct and/or incorrect decisions, rather than by precisely describing the decision process in its full generality and then transforming this description into a computer program.

A similar situation exists in computer chess. The development of computer programs playing at the master level (especially the end games) seems to be a formidable task if the programs are not eventually able to learn and improve on their decision making rules through the specific examples of games, rather than by being explicitly told all the rules.

Due to easy access to human knowledge about chess and the relative simplicity of testing the results, chess is one of the most attractive testing domains for inductive inference programs. This report presents first results from an experiment on the application of an inductive learning program called AQVAL/1 developed at the University of Illinois, to chess end games. The experiments were to infer from examples the classification rules which distinguish the win position from the draw position in the single pawn end game (i.e., white king and pawn against black king).

The problem of representing the human knowledge about this end game has been studied by Tan (Tan, 1972). The major and exceedingly significant result of his work is a comprehensive program which solves all known cases of this game (including all examples from Awerbach (Awerbach, 1958) and Fine (Fine, 1941).

His approach, however, has certain strong limitations which make it not very prospective for the more complex end games. As Tan (*loc. cit.*) admits himself:

> "The program does not have any capability of learning, modifying the advice or discovering useful predicates and action schemes. At the moment it is unable to explain its own moves or interpret the opponent's moves ('what is the threat?' 'what does he want?'). It also does not recognize similarity of positions to avoid repetition of analysis."

There has not yet been much done on the implementation of learning processes within the chess playing programs. An interesting effort in this direction has been reported recently by Pitrat (Pitrat, 1974).

In order to clarify the current goals of this work, it is useful to distinguish between different levels of learning, depending on what has to be given to a program learning decision making and what the program is able to do itself:

0-level: (Memorizing facts and rules)

The program has to know:

a) The representation space for the problem (Michalski, 1973), i.e., the program has to know which descriptors* to use in characterizing objects (e.g., situations in chess), and how to measure them,
b) a procedure (a decision rule) for computing the decision class (e.g., win or draw, good or bad move, etc. in chess) to be assigned to an object, based on the description of this object.

The program is able:

c) to memorize and execute the procedure to compute the decision class for any object based on its description.

(A learning process here involves simply the memorization of the methods of measuring descriptors and the decision algorithm.)

1-level: (Learning a decision rule from examples)

The program has to know:

a) the representation space (as in 0-level),
b) examples of objects of different decision classes and the decision classes they belong to.

The program is able:

c) to determine a generalized decision rule for computing a decision for any object,
d) to memorize and evaluate the decision rule for any object.

---

*A *descriptor* is a (unary or n-ary n=2,3,4, . . .) function from the objects or their parts into a set of 'descriptor values'. Unary descriptors are often called features.

2-level: (Learning descriptors and decision rules from examples)

The program has to know:

a) a partial representation space (i.e., to know some basic descriptors of the objects, not necessarily relevant to the specific problem, but which contain sufficient information for creating a new relevant representation space),
b) examples of objects and the correct decision classes they belong to.

The program is able:

c) to create a new problem-oriented representation space (e.g., 'discover' new descriptors) which is more adequate for the given family of decision classes,
d) to infer the generalized decision rules involving new descriptors,
e) to memorize and evaluate the inferred rule for computing decisions for any object.

One can distinguish further levels of learning but will stop here since they are not relevant to this paper. According to the above classification, Tan's work belongs to level 0. The work presented here belongs to level 1.

AQVAL PROGRAMS

AQVAL/1 programs constitute a package of programs for inductive inference and machine learning.

The following programs are presently in operation:

AQ7 — which infers an optimized description of one decision class in relation to other classes, based on given event sets. The program permits the user to define different optimization functionals, various modes of program operation and some other parameters.
(Michalski, 1973; Larson and Michalski, 1975)

AQ8 (Uniclass) — which determines an optimized description of each decision class separately, under the constraint that the 'degree of generalization' of the description will not exceed a certain value
(described in an internal report)

AQ9 — which optimizes a given set of DVL_1 formulas* according to a certain optimality functional
(Cuneo, 1975)

*DVL_1 formula stands for a disjunctive simple variable-valued logic formula (Michalski, 1975).

SYM-1 which determines symmetry (with regard to a set of variables) in variable-valued functions and creates DVL_1 formulas with symmetric selectors
(Jensen, 1975)

In the experiments reported here, we used only AQVAL/1–AQ7 program. The program is well described and documented (Michalski, 1973; Larson and Michalski, 1975) and therefore we will not discuss it here.

DESCRIPTION OF THE EXPERIMENTS

Relation to Tan's work

The end game is a part of the chess game for which a good deal of knowledge is available, and it is often possible to tell whether a move is correct or not. Like humans, machines should use that knowledge to play end games in contrast to the exclusive use of search. An important application of this idea is the program written by (Tan, 1972) to play the single-pawn end game (white king and white pawn against black king). In this program, knowledge is organized in a binary decision tree which associates values of predicates of the board positions with certain decisions. The non-terminal nodes of the tree correspond to predicates of the positions and the terminal nodes (leaves) correspond to ordered pairs <VALUE,ACTION>. VALUE indicates if the position is WINNING (whites win), DRAWING or UNDEFINED. ACTION determines the action scheme that leads to a correct move for that position. To play the game in a given board position the program goes through the decision tree until a terminal node is reached. In other words, the position is classified according to its predicates and the action scheme associated with its class is retrieved. If the VALUE of the terminal node is UNDEFINED, then the game tree is searched depth-first) until the first position that returns a value (winning for the whites and drawing for blacks) is reached. It is important to notice that the knowledge in this program is represented not only by the predicates of the positions but also by the order in which these predicates appear in the decision tree.

Due to the lack of learning capabilities and other drawbacks, Tan's approach is difficult to apply to more complicated end games. The organization of knowledge in a decision tree is by no means a simple task. It requires a great deal of knowledge of chess and even for a chess specialist it would be time consuming because there is no systematic way to do it. As Tan affirms, the organization of the decision tree for the single-pawn end-game was obtained through a trial and error process. It seems to us that the learning process defined by AQVAL could overcome this problem. The fact that the formulas generated by AQVAL programs can be made optimal or quasi-optimal according to given criteria (e.g., minimal number of terms*) imposes a natural order on the knowledge being

*i.e., products of selectors (Michalski, 1973). A simple form of a term is a conjunctive statement.

acquired. Another serious problem in Tan's program is the choice of adequate predicates. In a complicated end-game it will certainly not be easy to find the appropriate ones. Here again inductive learning could be used to generate predicates. A possibility for solving this particular problem is suggested in the end of the paper. Also, Tan's program is not a simple one in terms of the length and speed. A program to play an end-game using variable-valued logic formulas would be much simpler in those aspects because finding the value of a VL_1 formula for a specific event (chess position) is a very straightforward and simple operation.

The experiments described in the following sections represent the first step in testing these possibilities. It consists basically in generating variable-valued logic formulas for the single-pawn end game and testing the formulas. These were generated by inputting to AQ7 a set of learning events for each class of positions (winning or drawing). The formulas were tested by a different set of events. A testing event was assigned to one class if the percentage of terms that covered that event in the formula generated for that class was greater than the one in the formula obtained for the other class. In case of tie, the event was assigned to an undecided class.

First experiment

In the first experiment a sample of 242 learning events was used (120 winning positions and 122 drawing positions). These events were generated by taking examples from the classes of positions defined by the terminal nodes of the decision tree of Tan's program. The events were described by the following set of 17 variables (the numbers between the parenthesis define the range of the variables):

- x_1 TURN(0:1): 0-BLACK; 1-WHITE.
- x_2 PAWN CAN ADVANCE(0:1): If equal to 1, the pawn can move without being captured in the next move. That also implies that the pawn is not blocked.
- x_3 PAWN's FILE(0:1): 0-not rook pawn; 1-rook pawn.
- x_4 WHITE KING BLOCKS PAWN(0:1): If equal to 1 the white king is in front of the pawn.
- x_5 WHITE STALEMATE(0:1): If equal to 1 the white have no legal move.
- x_6 CAN CAPTURE(0:1): If equal to 1 the black king can capture the pawn in the present move. This variable has no meaning if it is white's turn.
- x_7 BLACK STALEMATE(0:1): If equal to 1 the black king has no legal move.
- x_8 PAWN CAN RUN(0:1): If equal to 1 the pawn can keep moving till it gets crowned without being intercepted by the black king.
- x_9 BLACK KING ON THE CORNER(0:1): If equal to 1 the black king is on an upper corner of the board.

x_{10} WHITE KING DOMINANT(0:1): If equal to 1 then we have that either of the 2 following cases occurs:

I. $5 \leqslant$ rank of pawn $\leqslant 7$ and rank of white king = rank of pawn + 1 and file of pawn $-1 \leqslant$ file of white king $\leqslant$ file of pawn + 1 and $2 \leqslant$ file of pawn $\leqslant 7$.

II. rank of white king = rank of pawn + 2 and file of pawn $- 1 \leqslant$ file of white king $\leqslant$ file of pawn + 1.

x_{11} BLACK KING AHEAD(0:1): If equal to 1 the black king blocks the pawn.

x_{12} PAWN'S RANK(0:3): 0-ranks 2, 3, and 4; 1-rank 5; 2-rank 6; 3-rank 7.

x_{13} WHITE KING'S RANK(0:5): 0-2 rows behind the pawn; 1-1 row behind the pawn, . . .; 5-3 rows in front of the pawn.

x_{14} WHITE KING'S FILE(0:3): 0-same as the pawn's file; 1-1 column from the pawn, . . .; 3-3 columns from the pawn.

x_{15} BLACK KING'S RANK(0:4): 0-1 row behind the pawn; . . .; 4-3 rows in front of the pawn.

x_{16} BLACK KING FILE'S(0:3): Similar to x_{14}.

x_{17} RELATIVE POSITION OF THE KINGS(0:1): 0-the kings are on the same side of the pawn, 1-the kings are on opposite sides of the pawn.

The first 11 variables correspond to some predicates used by Tan. The other variables, called positional variables, define the coordinates of the men on the board (it has been assumed, however, that only positions in which the kings are no more than 3 moves away from the pawn can be represented).

AQ7 was run twice. The learning events were presented in a different order each time. In the first run the events in the winning class were covered by 16 terms and the events in the drawing class by 18 terms. For the second run these number were 14 and 23 respectively. The formulas obtained are listed in the Appendix. The great majority of rules determined by the formulas were heavily dependent on the positional variables. The rules can easily be expressed in natural language. Here are some examples:

> "Black can draw the game when the pawn is a rook pawn and the black king is ahead of the pawn on the same column or on the adjacent one."
>
> "Black can draw the game when it's Black's turn, the pawn cannot advance and the white king does not block the pawn." (Actually there is one exception to this rule and that is the case in which the rank of the pawn is 7, the black king blocks the pawn and the white king is on the square immediately behind the pawn on its left or right side.)
>
> "White wins when it is White's turn, the pawn has a rank 7 and is not

a rook pawn, and the white king is on the square immediately behind the pawn and on the same column."

The formulas were tested with a sample of 159 events* (78 drawing positions and 81 winning positions). The testing events were generated in an analogous way to the learning events. The following table shows for each class of positions, the percentages of events classified correctly, incorrectly and not classified, for each formula separately and for both formulas considered as only one.

	Drawing Positions			Winning Positions		
	Correct	Incorrect	Undecided	Correct	Incorrect	Undecided
First Formulas	86%	5%	9%	86%	4%	10%
Second Formulas	83%	12%	5%	79%	12%	9%
Both Formulas Together	80%	4%	16%	80%	4%	16%

Second experiment

The second experiment consisted of two parts. In the first part, the formulas were obtained by running AQ7 with a set of 200 learning events† (97 drawing positions and 103 winning positions) randomly generated. In the second part, new formulas were generated by adding 54 additional events to the previous set of 200. These new 54 positions were obtained in the way described in the first experiment. The events were described by the following set of 15 variables (These variables are equal or similar to the variables used in the first experiment; variables WHITE STALEMATE or BLACK STALEMATE were not used here):

x_1 TURN(0:1): 0-BLACK, 1-WHITE. Same as x_1 in the first experiment.

x_2 PAWN'S RANK(0:5): 0-rank 2, . . ., 5-rank 7. The ranks 2, 3, and 4 that were clustered in the first experiment were separated here. Same as x_{12} in the first experiment.

x_3 PAWN'S FILE(0:1): 0-not rook pawn, 1-rook pawn. Same as x_3 in the first experiment.

x_4 WHITE KING'S RANK(0:6): 0-3 rows behind the pawn, 1-2 rows behind the pawn, . . ., 6-3 rows in front of the pawn. The variable

*The total number of events possible (the cardinality of the cartesian product of the domains of descriptors is approximately $8 \cdot 10^6$). It should be noted that this number is much larger than the number of possible board situations for which the upper-bound is only 64 x 63 x 62 = 250,000. The discrepancy is due to the fact that certain combinations of descriptor values correspond to impossible chess board situations.

†The size of event space here is approximately $5 \cdot 10^6$.

x_{13} of the first experiment was expanded to include 3 rows behind the pawn.

x_5 WHITE KING'S FILE(0:3): 0-same as the pawn's file, 1-1 column from the pawn, . . ., 3-3 columns from the pawn. Same as x_{14} in the first experiment.

x_6 BLACK KING'S RANK(0:6): 0-3 rows behind the pawn, 1-2 rows behind the pawn, . . ., 6-3 rows in front of the pawn. The variable x_{15} of the first experiment was expanded to include 3 rows behind the pawn.

x_7 BLACK KING'S FILE(0:3): 0-same as the pawn's file, 1-1 column from the pawn, . . ., 3-3 columns from the pawn. Same as x_{16} in the first experiment.

x_8 RELATIVE POSITION OF THE KINGS(0:1): 0-the kings are on the same side of the pawn, 1-the kings are on opposite sides of the pawn. Same as x_{17} in the first experiment.

x_9 BLACK KING ON THE CORNER(0:1): If equal to 1 the black king is on an upper corner of the board. Same as x_9 in the first experiment.

x_{10} PAWN CAN ADVANCE(0:1): If equal to 1 the pawn can move without being captured in the next move. That also implies that the pawn is not blocked. Same as x_2 in the first experiment.

x_{11} BLACK KING FAR FROM THE PAWN(0:1): If equal to 1 the black king is outside the region where it can intercept the pawn before it gets crowned by keeping moving. That does not mean that the pawn can run because it can be blocked by the white king. This variable is a generalization of the variable x_8 of the first experiment.

x_{12} WHITE KING BLOCKS PAWN(0:1): If equal to 1 the white king is in front of the pawn. Same as x_4 in the first experiment.

x_{13} BLACK KING IS AHEAD(0:1): If equal to 1 the black king is in front of the pawn. Same as x_{11} in the first experiment.

x_{14} WHITE KING IS ON CRITICAL SQUARES(0:1): If equal to 1 the position of the white king is defined by the following inequalities.

pawn's rank $<$ white king's rank $\leqslant$ pawn's rank + 2 pawn's file - 1 $\leqslant$ white king's file $\leqslant$ pawn's file + 1. This variable is a generalization of the variable x_{10} of the first experiment.

x_{15} PAWN UNPROTECTED(0:1): If equal to 1 than if it is black's turn the black kind can take the pawn in the next move. This variable is a generalization of the variable x_6 of the first experiment.

The formulas obtained for this experiment were again heavily dependent on the positional variables and are rather complicated. For the first part the events in the winning class were covered by 11 terms and the events on the drawing class were covered by 9 terms. For the second part these numbers were 23 and 23 respectively. The formulas are listed in the Appendix.

The formulas were tested with a sample of 200 events (86 drawing positions and 114 winning positions). The testing events were randomly generated. The results of the test are shown in the table below:

Learning Events	Drawing Positions			Winning Positions		
	Correct	Incorrect	Undecided	Correct	Incorrect	Undecided
200 random events	82%	8%	10%	92%	5%	3%
200 random events + 54 selected events	85%	6%	9%	83%	4%	13%

CONCLUSIONS AND SUGGESTIONS FOR FUTURE EXPERIMENTS

The results obtained in the two experiments described in this paper should be considered quite good if we realize that they represent the very first step in the learning process. The fact that the formulas obtained were heavily dependent on the lower-level positional variables seems to indicate that the other variables used (the ones corresponding to Tan's predicates) are not very significant in terms of the number of positions that they can cover. This was stressed in the second experiment when the events that are nicely described by these variables were added to the sample of randomly generated learning events and did not produce any significant improvement in the formulas. New variables should be introduced, probably ones corresponding to other predicates represented in Tan's program either through position patterns or, in a more complex way, through lookahead procedures that "test values of positions resulting from a hypothetical execution of an action scheme."

The next step in the chess-end-game experiment could be the improvement of the obtained formulas by feedback learning. This is a multi-step process that can be implemented in the following way:

1. Test the formulas with a sample of testing events. If all events are classified correctly stop, If not, go to step 2.
2. Obtain new formulas by inputting the formulas so far obtained to AQ9 program together with the events classified incorrectly in step 1. (AQ9 accepts events as well as terms as inputs.) Go to step 1.

This process should eventually produce quasi-correct formulas in the sense that all the events presented so far as inputs to the learning programs would be correctly classified by the formulas. The process can obviously be repeated ad nauseam with different samples of testing events. It is of great interest to see whether this process would eventually produce 'stable' decision rules and, also, how fast it could produce such rules, if it is possible to produce them in a reasonable time.

When dealing with more complicated end-games for which a nice set of descriptors (variables) is not available, it seems that some way of "discovering" new variables is necessary.

One possibility is to start generating formulas using exclusively lower-level variables (which describe only the positions of each man on the board). At the same time a set of position patterns is kept in the memory. These patterns could be described by some relations between the primitive variables, by array-images of the board, by geometric relations among some men on the board, or in any other convenient way. Every time that new learning events are input to AQVAL, some statistics are taken from them, so that each pattern will have associated with it the number of times that it occurred in each class of positions (winning, drawing, or losing). Whenever these numbers reach a certain threshold that shows that the corresponding pattern has some significance in separating the classes, the pattern is tentatively introduced as a new variable. If the use of that new variable simplifies the formulas so far obtained, it is kept with the other variables. If not, it is forgotten. These positional patterns can be at first introduced by humans. In a more advanced step they could be generated by the machine.

ACKNOWLEDGMENT

The authors would like to express their gratitude to Professor Donald Michie for the encouragement, useful comments, and interest in this work. This work was partially supported by the National Science Foundation under Grant No. DCR 74-03514.

REFERENCES

Awerbach, J., (1958) *Lehrbuch der Endspiele*, 1, Bauernendspile, Berlin.

Chernev, I. (1961) *Practical Chess Endings*, (Dover Publications, Inc., New York).

Cuneo, R.P. (1975) Selected Problems of Minimization of Variable-Valued Logic Formulas, M.S. Thesis, Department of Computer Science, University of Illinois.

Fine, R. (1941) *Basic Chess Endings* (David McKay Co., Inc., New York).

Horowitz, I.A. (1973) *How to Win in Chess Endings*, (David McKay Co., Inc., New York).

Jensen, G.M. (1975) SYM-1: A Program that Detects Symmetry of Variable-Valued Logic Functions, Department of Computer Science, University of Illinois.

Larson, J. and R.S. Michalski (1975) AQVAL/1 (AQ7): User's Guide and Program Description, Department of Computer Science, University of Illinois.

Michalski, R.S. (1973) Discovering Classification Rules by the Variable-Valued Logic System VL_1, Proc IJCAI3, Stanford, California.

Michalski, R.S. (1973) AQVAL/1—Computer Implementation of a Variable-Valued Logic System and the Application to Pattern Recognition," *Proceedings of the First International Joint Conference on Pattern Recognition*, Washington, D.C., October 30-November 1, 1973.

Michalski, R.S. (1974) Learning by Inductive Inference, NATO Advanced Study Institute on Computer Oriented Learning Processes, August 26-September 7, 1974, France (invited paper).

Michalski, R.S. (1975) Variable-Valued Logic and Its Applications to Pattern Recognition and Machine Learning, in *Multiple-Valued Logic and Computer Science*, (North-Holland Publishers).

Michie, D. A Theory of Evaluative Comments in Chess, *Proceedings of the AISB Summer Conference*, Brighton, University of Sussex.

Michie, D., A.P. Ambler, and R. Ross, Memory Mechanisms and Machine Learning, in *Simple Nervous Systems* (ed. B.R. Newth and P.W.R. Usherwood, London, Edward Arnold).

Newborn, M. (1975) *Computer Chess* (Academic Press, New York).

Pitrat, J. (1974) Realization of a Program Learning to Find Combinations in Chess, in *Computer Oriented Learning Processes*, NATO Advanced Study Institute, August 27-September 6, Bonas (gers) 1974.

Tan, S.T. (1972) Representation of Knowledge for Very Simple Endings in Chess, *Memorandum MIP-R-98*, School of Artificial Intelligence, University of Edingburgh.

APPENDIX

LIST OF THE FORMULAS GENERATED

The formulas obtained from VL_1 for the experiments are given in the tables below. Each line of a table represents a term of a VL_1 formula; each table is a complete formula, the disjunction of its terms.

Four values are tabulated for each term:

COV – Number of learning events covered by the term

NEW – Number of learning events covered by the term that were not covered by previous terms

IND – Number of learning events covered exclusively by this term

TOT – Number of learning events covered by all terms listed so far

VL_1 Formula	COV	NEW	IND	TOT
$[x_8=0]\ [x_{11}=0]\ [x_{12}=0,3]\ [x_{13}=0:3]\ [x_{14}=1,3]\ [x_{15}=1,3]\ [x_{16}=0,1]$ ∨	19	19	4	19
$[x_8=0]\ [x_{11}=0]\ [x_{12}=0,1,3]\ [x_{13}=0,1]\ [x_{15}=1,3,4]\ [x_{16}=0,1]\ [x_{17}=0]$ ∨	11	5	3	24
$[x_8=0]\ [x_{13}=1:3]\ [x_{14}=1,3]\ [x_{15}=2,3]\ [x_{16}=0]$ ∨	24	20	4	44
$[x_4=0]\ [x_8=0]\ [x_{12}=3]\ [x_{13}=0,3]\ [x_{14}=3]\ [x_{15}=0,2]$ ∨	3	2	1	46
$[x_1=0]\ [x_4=0]\ [x_8=0]\ [x_{13}=0,1,4]\ [x_{15}=2:4]\ [x_{16}=0,1]$ ∨	39	27	3	73
$[x_1=0]\ [x_4=0]\ [x_8=0]\ [x_{12}=0,3]\ [x_{13}=3]\ [x_{15}=1,4]$ ∨	3	3	0	76
$[x_3=1]\ [x_8=0]\ [x_{15}=2:4]\ [x_{16}=o,1]\ [x_{17}=0]$ ∨	43	17	13	93
$[x_{10}=0]\ [x_{11}=0]\ [x_{12}=0:3]\ [x_{13}=2,3]\ [x_{15}=1:3]\ [x_{16}=0:2]\ [x_{17}=0]$ ∨	18	7	4	100
$[x_3=1]\ [x_8=0]\ [x_{10}=0]\ [x_{12}=1,2]\ [x_{14}=0,2,3]\ [x_{15}=0,1]\ [x_{16}=0,1]\ [x_{17}=0]$ ∨	4	4	3	104
$[x_1=1]\ [x_3=0]\ [x_{12}=2]\ [x_{13}=2]\ [x_{14}=1]\ [x_{15}=3]\ [x_{16}=1]\ [x_{17}=0]$ ∨	2	2	2	106
$[x_1=0]\ [x_{12}=2]\ [x_{13}=3]\ [x_{14}=1]\ [x_{15}=3]\ [x_{16}=1]$ ∨	1	1	1	107
$[x_{10}=0]\ [x_{12}=0,2]\ [x_{13}=1,3,4]\ [x_{14}=0]\ [x_{15}=2:4]\ [x_{16}=0,2]$ ∨	14	6	6	113
$[x_1=0]\ [x_4=0]\ [x_8=0]\ [x_{14}=1]\ [x_{15}=0,3]\ [x_{16}=0,2]$ ∨	13	3	2	116
$[x_1=0]\ [x_3=1]\ [x_4=0]\ [x_8=0]\ [x_{12}=2]\ [x_{13}=2]$ ∨	5	1	1	117
$[x_4=0]\ [x_8=0]\ [x_{12}=0]\ [x_{13}=3]\ [x_{14}=1]\ [x_{15}=4]\ [x_{16}=1]$ ∨	1	1	1	118
$[x_2=0]\ [x_{13}=3]\ [x_{14}=2]\ [x_{15}=2]$ ∨	2	1	1	119
$[x_1=0]\ [x_2=0]\ [x_4=0]$ ∨	43	2	2	121
$[x_2=0]\ [x_3=0]\ [x_{12}=2]\ [x_{13}=4]\ [x_{14}=2]\ [x_{15}=3]\ [x_{17}=1]$	1	1	1	122

TABLE A1. First experiment, first run, drawing class

VL_1 Formula	COV	NEW	IND	TOT
$[x_3=0]\ [x_6=0]\ [x_{13}=1:4]\ [x_{14}=0:2]\ [x_{15}=0:2]$ V	51	51	22	51
$[x_1=1]\ [x_3=0]\ [x_9=0]\ [x_{11}=0]\ [x_{12}=2]\ [x_{13}=1,3\ 4]\ [x_{15}=3]\ [x_{16}=1:3]$ V	13	13	10	64
$[x_1=1]\ [x_{11}=0]\ [x_{12}=2,3]\ [x_{14}=1:3]\ [x_{16}=2,3]$ V	18	12	7	76
$[x_1=1]\ [x_5=0]\ [x_{11}=0]\ [x_{12}=0,1,3]\ [x_{13}=2:4]\ [x_{14}=0,2]\ [x_{15}=0,1,4]\ [x_{16}=1:3]$ V	21	9	5	85
$[x_1=0]\ [x_2=1]\ [x_{12}=2]\ [x_{13}=2]\ [x_{16}=1,3]$ V	5	3	3	88
$[x_6=0]\ [x_{11}=0]\ [x_{12}=1,2]\ [x_{13}=2:4]\ [x_{15}=0,4]\ [x_{16}=1,2]$ V	14	9	8	97
$[x_1=1]\ [x_{13}=0,2]\ [x_{14}=0,1]\ [x_{15}=2,4]$ V	7	5	5	102
$[x_1=1]\ [x_5=0]\ [x_{12}=1:3]\ [x_{13}=3,4]\ [x_{15}=0,1,4]\ [x_{16}=0,2]$ V	15	6	5	108
$[x_1=1]\ [x_6=0]\ [x_{13}=4,5]\ [x_{14}=1,3]\ [x_{15}=0,1]\ [x_{16}=1]$ V	4	2	2	110
$[x_3=0]\ [x_{12}=2,3]\ [x_{13}=0,2]\ [x_{14}=2]\ [x_{15}=2,3]$ V	7	5	4	115
$[x_2=1]\ [x_3=0]\ [x_{12}=0]\ [x_{13}=3]\ [x_{16}=0]$ V	1	1	1	116
$[x_1=1]\ [x_3=0]\ [x_{12}=3]\ [x_{13}=1]\ [x_{14}=0]$ V	1	1	1	117
$[x_1=1]\ [x_{12}=2]\ [x_{13}=2]\ [x_{17}=1]$ V	4	1	1	118
$[x_4=1]\ [x_{12}=1,3]\ [x_{16}=0,3]$	9	2	2	120

TABLE A2. First experiment, first run, winning class

VL_1 Formula	COV	NEW	IND	TOT
$[x_3=1]\ [x_{10}=0]\ [x_{13}=0,1,2,5]\ [x_{14}=0,1,3]\ [x_{15}=0:3]\ [x_{16}=0,1]\ [x_{17}=0]$ ∨	30	30	4	30
$[x_1=1]\ [x_{10}=0]\ [x_{12}=0,2,3]\ [x_{13}=2,3]\ [x_{14}=1,3]\ [x_{15}=1:3]\ [x_{16}=0,1]\ [x_{17}=0]$ ∨	10	8	2	38
$[x_1=0]\ [x_{10}=0]\ [x_{13}=0,1,4]\ [x_{14}=0,3]\ [x_{15}=2,3]$ ∨	16	8	1	46
$[x_2=0]\ [x_3=1]\ [x_{10}=0]\ [x_{13}=2,3]\ [x_{14}=0,2]\ [x_{15}=2]\ [x_{16}=0,2]$ ∨	8	8	4	54
$[x_2=0]\ [x_{10}=0]\ [x_{12}=0,3]\ [x_{13}=1:3]\ [x_{14}=1,3]$ ∨	20	10	6	64
$[x_4=0]\ [x_{12}=0:2]\ [x_{13}=2:4]\ [x_{14}=1,2]\ [x_{15}=2,3]\ [x_{16}=0]\ [x_{17}=0]$ ∨	16	9	2	73
$[x_{10}=0]\ [x_{12}=2]\ [x_{13}=4]\ [x_{14}=0]\ [x_{15}=2]\ [x_{16}=2]\ [x_{17}=0]$ ∨	1	1	1	74
$[x_3=1]\ [x_4=0]\ [x_8=0]\ [x_{10}=0]\ [x_{13}=1,4]\ [x_{14}=0,2]\ [x_{15}=0,1,2,4]\ [x_{16}=0,1]\ [x_{17}=0]$ ∨	11	5	1	79
$[x_4=0]\ [x_{10}=0]\ [x_{12}=0,3]\ [x_{13}=0,3]\ [x_{15}=1,4]\ [x_{16}=1]\ [x_{17}=0]$ ∨	9	6	3	85
$[x_2=0]\ [x_{10}=0]\ [x_{12}=0,2]\ [x_{15}=2:4]\ [x_{16}=0]\ [x_{17}=0]$ ∨	29	7	5	92
$[x_1=0]\ [x_{13}=1,3,4]\ [x_{15}=3]$ ∨	18	6	6	98
$[x_1=0]\ [x_{10}=0]\ [x_{12}=0,3]$ ∨	24	6	3	104
$[x_1=0]\ [x_3=0]\ [x_{12}=0,1]\ [x_{13}=2,4]\ [x_{14}=1]\ [x_{15}=0,3]\ [x_{16}=0,1]$ ∨	5	2	2	106
$[x_2=0]\ [x_{10}=0]\ [x_{13}=0]\ [x_{14}=3]\ [x_{15}=2]$ ∨	1	1	1	107
$[x_{10}=1]\ [x_{12}=3]\ [x_{13}=3]\ [x_{15}=1,2]\ [x_{16}=0,2]$ ∨	4	2	2	109
$[x_3=1]\ [x_{10}=0]\ [x_{14}=0]\ [x_{15}=3]$ ∨	6	2	2	111
$[x_2=0]\ [x_{10}=0]\ [x_{13}=4]\ [x_{14}=2]\ [x_{15}=3]\ [x_{17}=1]$ ∨	1	1	1	112
$[x_1=0]\ [x_{10}=0]\ [x_{14}=1]\ [x_{15}=3]\ [x_{16}=2]$ ∨	2	2	2	114
$[x_1=0]\ [x_3=1]\ [x_{14}=2]\ [x_{15}=3,4]$ ∨	5	3	3	117
$[x_2=1]\ [x_{13}=1]\ [x_{14}=1]\ [x_{15}=4]$ ∨	2	2	2	119
$[x_2=0]\ [x_{13}=0]\ [x_{14}=3]\ [x_{15}=1]$ ∨	2	1	1	120
$[x_1=0]\ [x_2=0]\ [x_4=0]\ [x_{13}=3]$ ∨	4	1	1	121
$[x_1=1]\ [x_3=1]\ [x_4=0]\ [x_{14}=2]\ [x_{16}=1]$	3	1	1	122

TABLE A3. First experiment, second run, drawing class

VL_1 Formula	COV	NEW	IND	TOT
$[x_1=1]\ [x_3=0]\ [x_5=0]\ [x_6=0]\ [x_{11}=0]\ [x_{12}=1:3]\ [x_{13}=2:4]\ [x_{14}=0:2]\ [x_{15}=0,1,2,4]$ V	34	34	11	34
$[x_1=1]\ [x_2=1]\ [x_6=0]\ [x_{12}=0,2,3]\ [x_{13}=0,1,2,4,5]\ [x_{14}=1:3]\ [x_{15}=0,1,2,4]$ V	22	18	10	52
$[x_2=1]\ [x_4=0]\ [x_5=0]\ [x_9=0]\ [x_{12}=1:3]\ [x_{13}=0,3]\ [x_{15}=0,2,3,4]\ [x_{16}=1,2]$ V	14	8	5	60
$[x_5=0]\ [x_6=0]\ [x_{11}=0]\ [x_{13}=1:3]\ [x_{14}=0:2]\ [x_{15}=0:2]\ [x_{16}=0,1,3]$ V	38	19	14	79
$[x_6=0]\ [x_{13}=3,4]\ [x_{14}=0]\ [x_{15}=0,4]\ [x_{16}=1,2]$ V	7	3	3	82
$[x_1=1]\ [x_3=0]\ [x_6=0]\ [x_{12}=0,3]\ [x_{13}=0,2,4]\ [x_{14}=0,2]\ [x_{15}=2]$ V	4	4	4	86
$[x_1=1]\ [x_3=0]\ [x_6=0]\ [x_7=0]\ [x_{12}=3]\ [x_{13}=1]\ [x_{14}=0]$ V	1	1	1	87
$[x_6=0]\ [x_{12}=1,2]\ [x_{13}=3]\ [x_{14}=0]\ [x_{15}=1,4]$ V	3	2	2	89
$[x_2=1]\ [x_{12}=1,2]\ [x_{13}=4]\ [x_{15}=0,1]\ [x_{16}=0,2,3]$ V	6	3	3	92
$[x_1=1]\ [x_2=1]\ [x_{12}=2]\ [x_{14}=1:3]\ [x_{15}=3]\ [x_{16}=2]$ V	7	7	6	99
$[x_1=1]\ [x_3=0]\ [x_9=0]\ [x_{11}=0]\ [x_{12}=2]\ [x_{13}=1,3,4]\ [x_{15}=3]\ [x_{16}=1,3]$ V	10	9	7	108
$[x_1=0]\ [x_3=0]\ [x_9=0]\ [x_{11}=0]\ [x_{12}=2]\ [x_{13}=2]\ [x_{16}=1]$ V	3	2	2	110
$[x_1=1]\ [x_3=0]\ [x_{12}=0,2]\ [x_{13}=2,3]\ [x_{17}=1]$ V	13	3	3	113
$[x_6=0]\ [x_{12}=1,3]\ [x_{13}=0,4]\ [x_{14}=1]\ [x_{15}=0,2]$ V	2	2	2	115
$[x_3=0]\ [x_{11}=0]\ [x_{12}=2]\ [x_{13}=1,2]\ [x_{14}=2]\ [x_{17}=0]$ V	7	4	4	119
$[x_1=1]\ [x_2=1]\ [x_{12}=0]\ [x_{16}=0]$	2	1	1	120

TABLE A4. First experiment, second run, winning class

VL_1 Formula	COV	NEW	IND	TOT
$[x_2=0:3]\ [x_4=0:2]\ [x_6=4:6]\ [x_7=0:2]\ [x_{14}=0]$ ∨	27	27	4	27
$[x_1=0]\ [x_6=2:6]\ [x_7=0,1]\ [x_8=0]\ [x_{14}=0]$ ∨	31	21	11	48
$[x_3=1]\ [x_6=3:6]\ [x_{11}=0]\ [x_{14}=0]$ ∨	48	27	16	75
$[x_1=0]\ [x_2=2:5]\ [x_4=1:6]\ [x_5=1:3]\ [x_6=2,3]\ [x_7=1,2]\ [x_{14}=0]$ ∨	12	10	10	85
$[x_1=0]\ [x_4=0,1]\ [x_6=4]\ [x_{14}=0]$ ∨	8	2	1	87
$[x_5=2,3]\ [x_6=4:6]\ [x_7=0,1]\ [x_{14}=0]$ ∨	22	4	3	91
$[x_2=1:4]\ [x_4=2]\ [x_5=2]\ [x_6=4:6]$ ∨	7	4	4	95
$[x_6=5]\ [x_7=2]\ [x_8=0]\ [x_{12}=0]$ ∨	3	1	1	96
$[x_2=5]\ [x_4=0]\ [x_6=4]$	3	1	1	97

TABLE A5. Second experiment, first part, drawing class

VL_1 Formula	COV	NEW	IND	TOT
$[x_4=1:5]\ [x_5=0:2]\ [x_6=0:2]\ [x_8=0]\ [x_{11}=1]\ [x_{13}=0]\ [x_{15}=0]$ ∨	35	35	5	35
$[x_3=0]\ [x_4=2:4]\ [x_5=0,1]\ [x_6=1:5]\ [x_{13}=0]\ [x_{15}=0]$ ∨	17	11	7	46
$[x_2=1,2]\ [x_4=3:5]\ [x_7=3]\ [x_{10}=1]\ [x_{15}=0]$ ∨	11	8	4	54
$[x_4=2:6]\ [x_6=0,1]\ [x_{10}=1]\ [x_{11}=1]\ [x_{15}=0]$ ∨	37	17	5	71
$[x_1=1]\ [x_6=1:4]\ [x_{11}=1]\ [x_{13}=0]$ ∨	38	18	12	89
$[x_3=0]\ [x_4=3:6]\ [x_5=1,2]\ [x_6=3:6]\ [x_7=1:3]\ [x_8=1]\ [x_9=0]$ ∨	10	5	5	94
$[x_2=4]\ [x_4=0,1]\ [x_6=0:4]\ [x_7=2,3]\ [x_8=0]\ [x_{13}=0]$ ∨	7	3	3	97
$[x_3=1]\ [x_4=4,5]\ [x_6=3,4]\ [x_7=2,3]\ [x_{11}=1]\ [x_{13}=0]$ ∨	2	2	2	99
$[x_2=5]\ [x_4=2]\ [x_6=4]\ [x_7=2]\ [x_{13}=0]$ ∨	2	1	1	100
$[x_2=0,1]\ [x_4=6]\ [x_6=2,4]\ [x_7=3]\ [x_{13}=0]$ ∨	2	2	2	102
$[x_6=1]$	25	1	1	103

TABLE A6. Second experiment, first part, winning class

VL_1 Formula	COV	NEW	IND	TOT
$[x_2=2:5]\ [x_4=4]\ [x_5=2,3]\ [x_6=2:6]\ [x_7=0:2]\ [x_{14}=0]\ [x_{15}=0]$ V	7	7	3	7
$[x_2=0:3]\ [x_4=0:2]\ [x_7=0:2]\ [x_8=0]\ [x_{11}=0]\ [x_{14}=0]$ V	30	30	13	37
$[x_1=0]\ [x_2=1:4]\ [x_4=1:6]\ [x_5=1:3]\ [x_6=2:6]\ [x_8=0]\ [x_{14}=0]$ V	32	22	7	59
$[x_2=1:5]\ [x_3=1]\ [x_5=1:3]\ [x_6=2,3]\ [x_7=0:2]\ [x_8=0]\ [x_{14}=0]$ V	15	7	3	66
$[x_4=1:3]\ [x_5=2,3]\ [x_6=6]\ [x_7=2]\ [x_{11}=0]\ [x_{14}=0]$ V	4	3	1	69
$[x_2=0,1]\ [x_4=3:6]\ [x_6=4:6]\ [x_7=0:2]\ [x_8=0]\ [x_{11}=0]\ [x_{12}=0]$ V	14	10	8	79
$[x_2=2:4]\ [x_5=0:2]\ [x_6=4,5]\ [x_7=0]\ [x_{10}=0]\ [x_{11}=0]\ [x_{12}=0]$ V	8	5	2	84
$[x_1=0]\ [x_2=2:5]\ [x_4=0:2]\ [x_5=0,1]\ [x_6=4,5]\ [x_{11}=0]\ [x_{12}=0]$ V	10	5	3	89
$[x_5=3]\ [x_6=4,5]\ [x_7=0,1]\ [x_{10}=0]\ [x_{11}=0]$ V	6	4	2	93
$[x_1=0]\ [x_2=3]\ [x_4=1:5]\ [x_5=1,2]\ [x_6=2:5]\ [x_7=1,2]\ [x_{15}=0]$ V	5	2	1	95
$[x_1=0]\ [x_2=1:3]\ [x_4=5,6]\ [x_6=2,3]\ [x_7=1,2]\ [x_{12}=0]$ V	5	3	1	98
$[x_2=2]\ [x_4=4:6]\ [x_5=1,2]\ [x_6=6]\ [x_8=0]\ [x_{11}=0]$ V	2	2	1	100
$[x_1=0]\ [x_6=2,3]\ [x_7=1]\ [x_{12}=0]$ V	10	4	4	104
$[x_2=5]\ [x_4=2]\ [x_6=2]\ [x_7=2]\ [x_{15}=0]$ V	1	1	1	105
$[x_2=1:3]\ [x_4=2]\ [x_5=2]\ [x_6=5,6]\ [x_8=0]\ [x_{11}=0]$ V	3	2	1	107
$[x_2=1:5]\ [x_3=1]\ [x_4=1:4]\ [x_6=3:5]\ [x_8=0]\ [x_{11}=0]$ V	27	5	3	112
$[x_2=5]\ [x_3=0]\ [x_4=0]\ [x_5=0,1]\ [x_6=4]\ [x_7=0:2]\ [x_8=0]$ V	3	2	2	114
$[x_1=0]\ [x_2=1:4]\ [x_4=5,6]\ [x_6=2:5]\ [x_7=0,1]\ [x_{12}=0]$ V	8	2	2	116
$[x_1=1]\ [x_3=1]\ [x_4=3,4]\ [x_6=3:5]$ V	9	2	2	118
$[x_1=0]\ [x_2=5]\ [x_4=1]\ [x_6=4]\ [x_7=2]\ [x_8=0]$ V	1	1	1	119
$[x_1=0]\ [x_2=1,2]\ [x_6=6]\ [x_{11}=0]$ V	9	2	2	121
$[x_1=1]\ [x_4=4]\ [x_6=3]\ [x_8=0]\ [x_{11}=0]$ V	2	1	1	122
$[x_1=1]\ [x_2=1]\ [x_6=3]$	2	1	1	123

TABLE A7. Second experiment, second part, drawing class

VL_1 Formula	COV	NEW	IND	TOT
$[x_4=2:5]\ [x_5=0:2]\ [x_6=0:3]\ [x_{10}=1]\ [x_{11}=1]\ [x_{13}=0]\ [x_{15}=0]$ v	37	37	14	37
$[x_1=1]\ [x_3=0]\ [x_5=0:2]\ [x_6=0:4]\ [x_{13}=0]$ v	35	26	9	63
$[x_2=2]\ [x_4=4,5]\ [x_6=0,1]\ [x_7=2,3]\ [x_8=0]\ [x_{13}=0]\ [x_{15}=0]$ v	2	2	0	65
$[x_2=0,1]\ [x_4=4,5]\ [x_7=3]\ [x_{13}=0]\ [x_{15}=0]$ v	6	5	2	70
$[x_2=4,5]\ [x_3=0]\ [x_4=1:3]\ [x_5=1,2]\ [x_7=0]\ [x_{10}=1]\ [x_{13}=0]$ v	3	2	1	72
$[x_2=4]\ [x_3=0]\ [x_4=0:4]\ [x_5=0]\ [x_7=2,3]\ [x_{13}=0]\ [x_{15}=0]$ v	6	2	1	74
$[x_2=0:3]\ [x_3=0]\ [x_4=3:6]\ [x_6=4:6]\ [x_7=1:3]\ [x_8=1]\ [x_{10}=1]\ [x_{13}=0]\ [x_{15}=0]$ v	12	8	7	82
$[x_1=1]\ [x_2=2:5]\ [x_{10}=1]\ [x_{11}=1]$ v	34	14	6	96
$[x_2=2,3]\ [x_4=3:6]\ [x_6=0,1]\ [x_{11}=1]\ [x_{13}=0]\ [x_{15}=0]$ v	17	5	5	101
$[x_2=1,2]\ [x_3=0]\ [x_4=2,3]\ [x_5=0,1]\ [x_{13}=0]$ v	4	2	1	103
$[x_2=2:4]\ [x_3=0]\ [x_4=4]\ [x_5=0,1]\ [x_6=2:5]\ [x_7=1:3]\ [x_{13}=0]\ [x_{15}=0]$ v	6	3	2	106
$[x_2=0,1]\ [x_3=0]\ [x_4=4:6]\ [x_5=0,1]\ [x_7=1:3]\ [x_{11}=1]\ [x_{13}=0]\ [x_{15}=0]$ v	6	3	3	109
$[x_2=0]\ [x_3=1]\ [x_4=3]\ [x_7=0]\ [x_{13}=0]$ v	1	1	1	110
$[x_2=3,4]\ [x_4=4,5]\ [x_5=0,1]\ [x_8=0]\ [x_{13}=0]\ [x_{15}=0]$ v	11	4	4	114
$[x_1=0]\ [x_2=3,4]\ [x_3=1]\ [x_4=2,3]\ [x_5=3]\ [x_{10}=1]\ [x_{13}=0]$ v	2	2	2	116
$[x_1=1]\ [x_4=5,6]\ [x_5=1,2]\ [x_6=3:5]\ [x_7=1,2]\ [x_{13}=0]\ [x_{15}=0]$ v	7	4	2	120
$[x_2=1]\ [x_3=0]\ [x_4=1]\ [x_5=0]\ [x_6=2]\ [x_7=3]\ [x_8=0]\ [x_{13}=0]$ v	1	1	1	121
$[x_1=1]\ [x_2=1:3]\ [x_4=3:5]\ [x_5=0,1]\ [x_7=0,1]\ [x_8=0]\ [x_{13}=1]$ v	5	2	2	123
$[x_2=2:4]\ [x_3=0]\ [x_4=2]\ [x_5=2,3]\ [x_7=1]\ [x_{15}=0]$ v	2	2	1	125
$[x_2=5]\ [x_3=1]\ [x_4=2]\ [x_8=0]\ [x_{15}=0]$ v	6	1	1	126
$[x_2=5]\ [x_6=1]\ [x_8=0]\ [x_{15}=0]$ v	4	1	1	127
$[x_1=1]\ [x_2=5]\ [x_4=4]\ [x_6=2]\ [x_{15}=0]$ v	1	1	1	128
$[x_1=1]\ [x_2=4]\ [x_3=0]\ [x_{15}=0]$	24	3	3	131

TABLE A8. Second experiment, second part, winning class

11

Inductive Learning in a Hierarchical Model for Representing Knowledge in Chess End Games

Pericles Negri

Department of Computer Science
University of Illinois at Urbana-Champaign

INTRODUCTION

The end game is the part of the chess game for which, in contrast with the other parts of the game, a good deal of knowledge is available that permits the player to find the correct move, if it exists. The problem then is to define precisely what this knowledge is, how it is organized and how it could be implemented in a computer so that the machine can play the end games efficiently without recurring to brute force searches.

We present here a model for this knowledge that is based on the concept of goals and subgoals. The primary goal of either White or Black is to win or to draw the game. This fundamental goal can be expressed as a logical function of some position predicates that represent geometrical or logical relationships among the men on the board, and some simpler subgoals that can be of offensive or defensive nature. In their turn these subgoals can be expressed as a function of their own subgoals and so on until very simple subgoals are reached that can be expressed exclusively as a function of primitive position predicates. To play the game in a determined position, the player tries to achieve some combinations of the subgoals of the primary goal determined by the logical function. This is done by trying to achieve simpler subgoals and so on through the very simple subgoals of the bottom of the structure. Whenever the last ones fail to be achieved, as characterized by the position predicates, the player goes back in a bottom-up direction and tries to achieve alternative subgoals.

Attached to the functions that define the primary goal in terms of its subgoals, there are special procedures that generate the move in case some combinations of these subgoals can be achieved.

Finally, it should be emphasized that the depth of the structure is quite arbitrary and different players will likely have different models for the same end game. Experienced players have probably less deep structures than unexperienced players because they eliminate some intermediate subgoals (what makes the functions more complex).

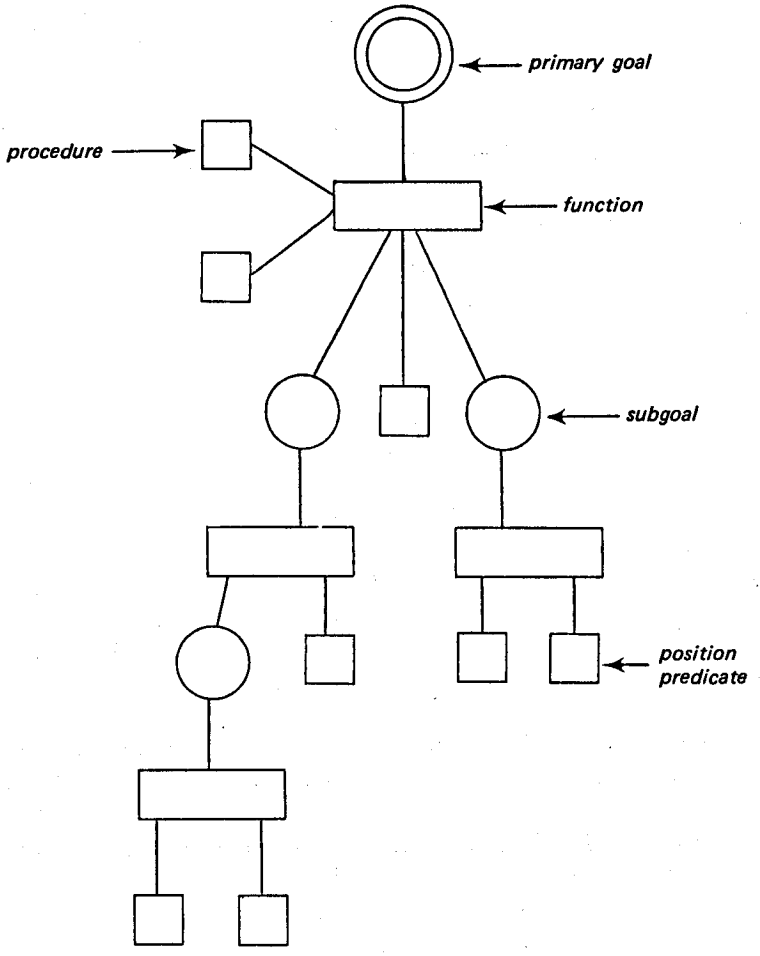


FIG. 1. A schematic representation of the knowledge in a hypothetical chess end game

Figure 1 is a schematic representation of the structure proposed but it should be noticed that the function blocks hide the fact that when a goal can be achieved through different combinations of its subgoals, some combinations will be more efficient than others, which implies a priority structure within those blocks.

LEARNING IN THE HIERARCHICAL STRUCTURE

Description of the model: single-pawn game case

In order to implement the proposed structure for chess end games in a computer we must be able to specify the subgoals, the position predicates and the functions that express a subgoal in terms of its subgoals and position predicates.

The latter could be a very difficult task unless some systematic way of doing it can be devised. Inductive learning through a variable-valued logic system like VL1 offers a solution for the problem. In order to test this, a partial model for the single-pawn end game (white king and pawn against black king) was constructed and VL1 formulas were obtained for two function blocks of the model.

To express a goal in terms of its subgoals and position predicates the following approach was used:

(a) Assign to each offensive subgoal a variable that can assume 3 values:

0-the subgoal cannot be achieved
1-the subgoal can be achieved
2-the subgoal was already achieved

(b) Assign to each defensive subgoal a variable that can assume 3 values:

0-the subgoal cannot be achieved (although it should)
1-the subgoal must be achieved (the opponent makes a threat).
2-the subgoal does not need to be achieved (the opponent is not making any threat)

(c) Assign to each position predicate a variable that assume as many values as the number of states of the predicate.

(d) Generate a learning sample of positions for the classes defined by each value of the variable that represents the goal for which the formulas are to be generated.

(e) Run the programs of the VL1 system (Larson, Michalski, 1975).

(f) Generate a testing sample of positions for the classes defined by each value of the variable for which the formulas were generated. Test the formulas obtained in the item (e) with the testing positions. Add the positions misclassified by the formulas to the learning sample and rerun the VL1 system programs. Keep doing this till all the testing positions are classified correctly by the formulas.

The partial model of the single-pawn end game consists of 4 submodels:

-Rook Pawn, Black's turn
-Rook Pawn, White's turn
-Non Rook Pawn, Black's turn
-Non Rook Pawn, White's turn

- *Rook pawn, Black's turn*

Primary Goal: To draw the game. It's a function of three subgoals and one position predicate (the pawn can run through the 8th row and get crowned).

Subgoals of the Primary Goal:

1-to take the pawn

2-to enter the critical squares (figure 2). Once inside the critical squares the black king can keep inside them and avoid the pawn getting crowned.

3-to reach the critical position (figure 3). From the critical position the black king can either take the pawn or enter the critical squares.

The subgoal "to take the pawn" can be expressed in terms of simpler subgoals or directly in terms of position predicates. The other two can be expressed directly in terms of the position predicates.

- *Rook pawn, White's turn*

Primary Goal: To crown the pawn. It can be expressed in terms of a position predicate (the pawn can run) and two subgoals.

Subgoals of the Primary Goal:

1-to defend the critical squares.

2-to defend the pawn

These subgoals can be expressed directly in terms of position predicates.

- *Non-rook pawn, White's turn*

Primary Goal: To crown the pawn. It can be expressed as a function of one position predicate (the pawn can run) and four subgoals.

Subgoals of the Primary Goal:

1-to enter the critical squares of type 1 (figure 4).

2-to enter the critical squares of type 2 (figure 5) when the rank of the pawn is greater than 5.

3-to defend the pawn.

4-to enter the critical position (figure 6) when the rank of the pawn is equal to 7.

There is one exception to the subgoals 1 and 2 when the pawn is the knight pawn, it has rank equal to 6 and the black king is on the corner of the board. All these subgoals can be expressed directly in terms of position predicates.

- *Non-rook pawn, Black's turn*

Primary Goal: To draw the game. It can be expressed as a function of two position predicates (pawn can run, stalemate position) and five subgoals.

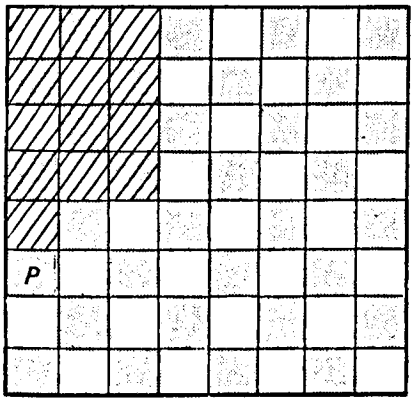


FIG. 2. Critical squares in the rook-pawn case.

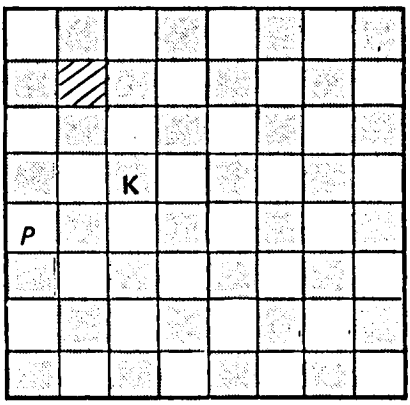


FIG. 3. Critical position in the rook-pawn case; the white king cannot be in the shadowed square

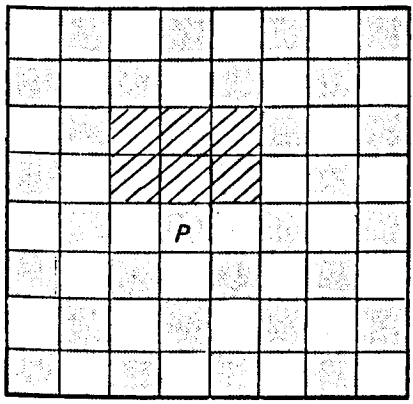


FIG. 4. Critical squares of type 1 for white king in the non-rook-pawn case

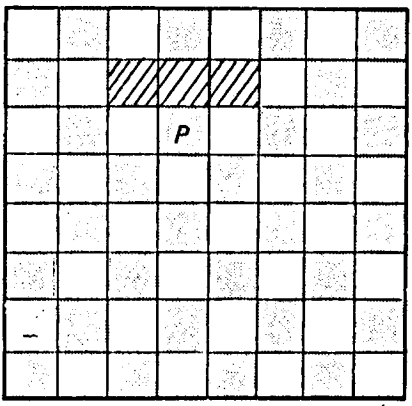


FIG. 5. Critical squares of type 2 for white king in the non-rook-pawn case

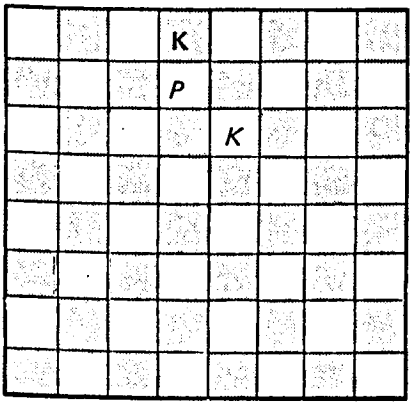


FIG. 6. Critical position for white in the non-rook-pawn case

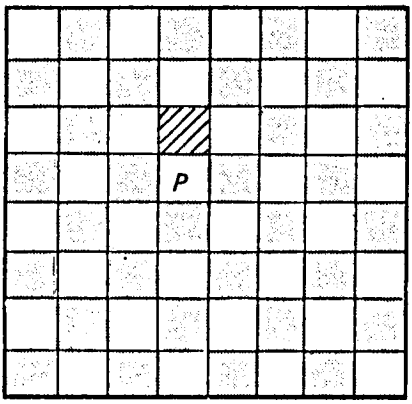


FIG. 7. Square 1 for the black king in the non-rook-pawn case

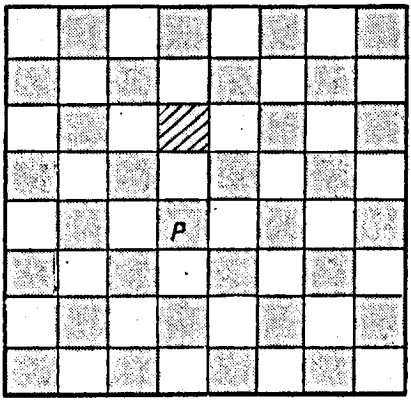


FIG. 8. Square 2 for the black king in the non-rook-pawn case

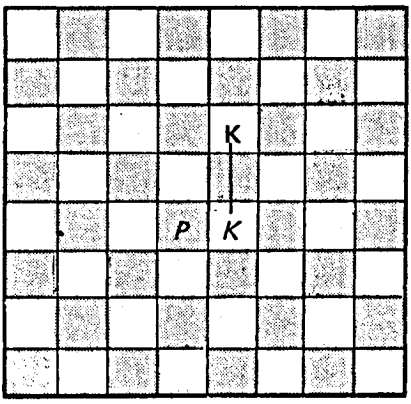


FIG. 9. Opposition of type 1

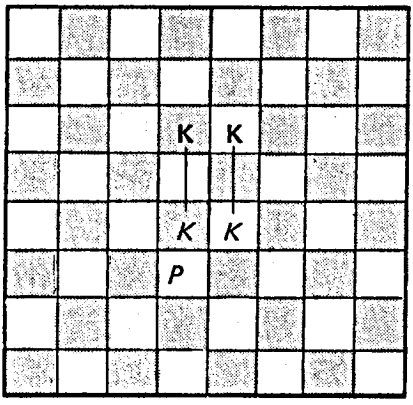


FIG. 10. Opposition of type 2

Subgoals of the Primary Goal:

1-to take the pawn
2-to reach square 1 (figure 7).
3-to reach square 2 (figure 8) when the rank of the pawn is less than 6.
4-to enter in opposition of type 1 (figure 9).
5-to enter in opposition of type 2 (figure 10) when the rank of the pawn is less than 5.

These subgoals can be expressed in terms of position predicates.

Learning experiments

First experiment

The first experiment was to generate the formulas that define the primary

goal of the black (rook pawn case) as a function of its subgoals and position predicate.

Variables used:

X1: 0-the pawn cannot run
1-the pawn can run

X2: 0-black king cannot enter the critical squares.
1-black king can enter the critical squares.
2-black king already in the critical squares.

X3: 0-black king cannot take the pawn.
1-black king can take the pawn.

X4: 0-black king cannot reach the critical position.
1-black king can enter the critical position.
2-black king already in the critical position.

The formulas obtained have the following terms:

Class 1: Black can draw the game

$$(X1=0)\,(X4=1,2)\,v$$
$$(X3=1)\,v$$
$$(X1=0)\,(X2=1,2)$$

Class 2: Black cannot draw the game

$$(X1=1)\,v$$
$$(X2=0)\,(X3=0)\,(X4=0)$$

This experiment was very simple and it was done only to illustrate the learning process for a high level of the structure. The learning positions were chosen in order to cover the whole event space. This could be done only because the number of variables is very small as well as their cardinalities. The formulas were obtained in one step of the learning process and as they are self evident they were not tested.

Second experiment

The second experiment was done to illustrate the learning process in a low level of the structure in which a subgoal is expressed exclusively as a function of position predicates. The formulas were obtained for the subgoal "black king can take pawn" in the rook pawn case.

Variables used:

X1: 0-black king in the region A of the board. (figure 11).
1- " " " " " B " " " "
2- " " " " " C " " " "
3- " " " " " D " " " "
4- " " " " " E " " " "

a	a	a	a	b	c	c	c
a	a	a	b	c	c	c	c
a	a	b	c	c	c	c	c
a	b	c	c	c	c	c	c
(P)	c	c	c	c	c	c	c
e	d	d	d	d	d	d	d
e	e	d	d	d	d	d	d
e	e	e	d	d	d	d	d

FIG. 11. Division of the board into regions according to the pawn position

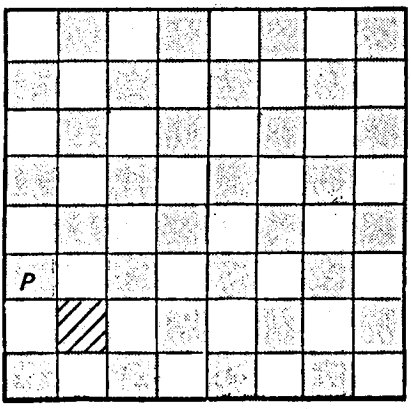


FIG. 12. Square *a*

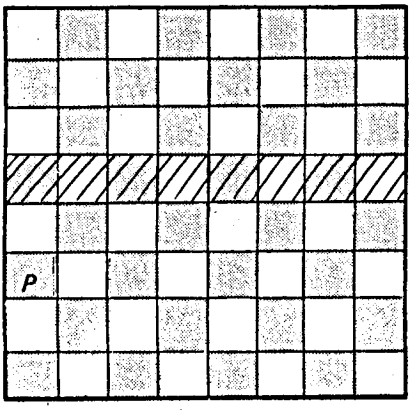


FIG. 13. Row *a*

X2: 0-white king in the region A of the board.

1- " " " " " B " " "
2- " " " " " C " " "
3- " " " " " D " " "
4- " " " " " E " " "

X3: 0-distance black king-pawn - distance white king-pawn <-2

1- " " " " " " " " =-1
2- " " " " " " " " =-1
3- " " " " " " " " = 0
4- " " " " " " " " = 1
5- " " " " " " " " >1

X4: 0-distance black king-pawn - distance white king-square *a* < -1
1- " " " " " " " " " " = -1
2- " " " " " " " " " " = 0
3- " " " " " " " " " " = 1
4- " " " " " " " " " " > 1
See figure 12 for the square *a*.

X5: 0-distance black king-pawn's file - distance white king-pawn's file < -1

X5: 0-distance black king-pawn's file - distance white king-pawn's file < -1
1- " " " " " " " " " " = -1
2- " " " " " " " " " " = 0
3- " " " " " " " " " " = 1
4- " " " " " " " " " " > 1

X6: 0-distance black king-pawn's rank - distance white king-pawn's rank < -1
1- " " " " " " " " " " = -1
2- " " " " " " " " " " = 0
3- " " " " " " " " " " = 1
4- " " " " " " " " " " = 1

X7: 0-distance black king-row *a* - distance white king-row *a* < -1
1- " " " " " " " " " " = -1
2- " " " " " " " " " " = 0
3- " " " " " " " " " " = 1
4- " " " " " " " " " " > 1
See figure 13 for row *a*.

X8: 0-pawn cannot run
1-pawn can run

X9: 0-pawn's rank = 2
1-pawn's rank = 2

X10: 0-black king cannot take the pawn in the present move
1-black king can take the pawn in the present move

The formulas obtained have the following terms:

Class 1: The black king can take the pawn

(X1=0) (X3=0:3) (X4=0) (X8=0) (X9=1) ∨
(X4=0:2) (X5=0) (X8=0) ∨
(X3=0:3) (X6=0,1) (X7=0) (X8=0) (X9=1) ∨
(X1=2:4) (X3=0:2) (X6=0:3) (X8=0) (X9=1) ∨

(X1=2:4) (X3=3,4) (X5=3) (X6=0,1) (X7=0:3) (X8=0) v
(X1=2,3) (X2=1:4) (X3=0:3) (X4=0:2) (X8=0) (X9=1) v
(X3=0,1) (X5=2:4) (X6=0) (X8=0) v
(X1=2) (X4=0,1) (X5=0:3) (X7=0,1) v
(X3=1,2) (X6=0) (X7=3,4) (X8=0) v
(X3=3) (X4=2) (X7=0,1) v
(X1=0) (X3=2)

Class 2: The black king cannot take the pawn

(X3=3:5) (X5=1:4) (X6=2:4) (X9=1) (X10=0) v
(X1=0:2) (X2=0:2) (X4=3,4) (X5=0:2) (X9=1) v
(X1=1:4) (X2=0,1) (X3=2:5) (X6=1:4) (X10=0) v
(X1=2) (X4=1:4) (X5=3,4) (X6=0) (X7=3,4) (X9=1) v
(X1:4) (X4=1:4) (X8=1) v
(X1=0,1) (X4=1:4) (X5=1:4) (X6=2:4) v
(X1=1:4) (X3=4,5) (X5=4) (X7=0:3) v
(X1=3,4) (X2=0) (X5=3) (X7=1:4) (X10=0) v
(X1=0:3) (X6=0) (X7=2) (X9=0) v
(X1=2:4) (X3=2) (X5=4) (X7=0) v
(X1=1:4) (X3=1:5) (X6=4) (X9=0) v
(X4=3,4) (X5=0,1).
(X8=1)

Initial samples of 120 learning positions and 120 testing positions were used. After three steps of adjusting the formulas by adding to the learning sample the testing positions that were misclassified we got all the testing positions correctly classified by the last formulas. The learning positions were generated by dividing the positions in 25 classes defined by the 25 possible combinations of the values of the variables X1 and X2 and picking from each class some positions according to a 'near miss' criterion (Michalski, 1975) in which positions which are near to a 'critical line' that separates the class 1 from class 2 are taken. The testing positions were generated by taking some general positions from the 25 classes defined before.

CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

In a previous work (Michalski and Negri, this volume) we tried to introduce inductive learning in chess end games by generating formulas for two classes of positions of the single-pawn end game: winning positions (for White) and drawing positions (for Black). The variables used were position predicates used by Tan (Tan, 1972) in his model for this end game and other variables that represented the positions of the men on the board. The results although not totally unsatisfactory showed the deficiencies of the model used: the formulas obtained were heavily dependent on the second type of variable. This indicates that the variables of the first type were too specific to some groups of positions so the

"preference" for the second type of variables that are more general. What happened is that we clustered in two classes positions that had quite different characteristics. To get simple formulas for these classes we would have to use a large number of variables to cover all those characteristics, which is not practical.

In the model that we propose in this work, we use the learning process to generate formulas for some reduced classes of positions, namely those associated with some subgoals of the structure. This permits us to use exclusively variables that are relevant for those classes of positions in particular, which makes the learning process simpler and more efficient.

We think that the hierarchical structure defined by our model is very close to the way people play chess end games. The procedures attached to the structure to generate moves can be very simple because they generate moves that have to reach some determined goal. A simple search process can be efficiently used because we can eliminate *a priori* from the search moves that will not lead to the desired goal. This seems to be the kind of process involved in human thinking that automatically rejects irrelevant moves. Besides that, information obtained in the lower levels of the structure can be used by those procedures.

We did not try to develop a playing model for the case in which the desired goals cannot be achieved, i.e., the position is a losing one. But we think it could be done because, at least we can know from the formulas the "weaknesses" of the opponent and try to set traps for him.

The next research step should be the development of automatic ways of generating the position predicates that are used in the bottom of the structure for defining the simple subgoals. This is boring work and it would be nice if it could be automated. A possibility could be to extend to chess the ideas developed by Newman and Uhr (Newman and Uhr, 1965) for board games like Go Moku and Tic Tac Toe. Their method consists in defining a utility index for the patterns that appear on the board during the game. This can be defined simply as the number of times the pattern appeared in winning positions over the number of times the pattern appeared at all. Patterns that have roughly the same indexes are put in the same class. Now to the patterns that belong to some class with a high index of utility, geometric transformations like rotation, translation and reflexion are applied in order to transform several of these patterns into a stereotype pattern. The patterns that are considered equivalent under those transformations define a "predicate" of the positions. The problem in extending these ideas to chess is that as Pitrat observed (Pitrat, 1974), the logical relationships among the men on the board are not easily expressed by geometrical transformations. But if we use VL1 formulas to cover the positions with the same utility index against the other positions, the formulas so generated could define the desired predicates. In this case these formulas would use exclusively primitive variables like position descriptors for the men on the board. Finally it should be noticed that these formulas should use the full power of the VL1 language in order to include selectors of the kind $[X_i\text{-}X_j<K]$, where K is some constant. The generation of formulas for the variables used in the second

experiment described in this paper is a clear example for the necessity of this kind of selector. Programs like Jensen's (1975) for synthesizing VL_1 formulas with symmetric selectors could be used for that purpose.

REFERENCES

Jensen, G.M. (1975) SYN-1: A Program that Detects Symmetry of Variable-Valued Logic Functions. Department of Computer Science, University of Illinois.

Larson, J. and Michalski, R.S. (1975) AQVAL/1 (AQ7): User's Guide and Program Description. Department of Computer Science, University of Illinois.

Michalski, R.S. (1973) AQVAL/1–Computer Implementation of a Variable-Valued Logic System and the Application to Pattern Recognition, *Proceedings of the First International Joint Conference on Pattern Recognition*, Washington, D.C.

Michalski, R.S. (1975) On the Selection of Representative Samples from Large Relational Tables for Inductive Inference. M.D.C. 1.1.9 Department of Information Engineering, University of Illinois at Chicago Circle.

Newman, C. and Uhr, L. (1965) BOGART: A Discovery and Induction Program for Games. *ACM 20th National Conference.*

Pitrat, J. (1974) Realization of a Program Learning to Find Combinations in Chess. *Computer Oriented Learning Processes.* NATO Advanced Study Institute, Bonas (gers) 1974.

Tan, S.T. (1972) Representation of knowledge for very simple endings in chess, *Memorandum MIP-R-98,* School of Artificial Intelligence, University of Edinburgh.

12

Summing up of the Discussion on Inductive Inference

I. J. Good

Virginia Polytechnic Institute and State University
Blacksburg, Virginia

A marriage of the three approaches to induction described in this colloquium might be fruitful. Michalski's inductive hypotheses are subject to no exceptions, whereas Buchanan and his coworkers are prepared to throw out "strays" or "outliers." Perhaps it would ultimately be useful to integrate these approaches by using the probabilistic notion of explicativity. For this should enable us to take an explicit attitude to each hypothesis intermediate between acceptance and rejection as people often do in real life.

Michalski's use of logic is analogous to what hardware designers call "logic-circuits." This is not surprising since he has an engineering background. The analogy is very close because the choice of primitive logical operations is necessarily based on economic principles in some sense. Perhaps the human brain also contains an economical collection of built-in logic circuits. Certainly the experiments of Hubel and Weisel have partially proved that this is so for the cat.

It would be interesting to know the whole history of the logical approach to hypothesis formulation. My knowledge of the literature is not good, but I can cite an example from my own work (Good, 1959, p. 25), (Good, 1967, p. 94). It was pointed out there that the optimization of the coefficients of a linear evaluation function can hardly be regarded as concept formation, but as soon as quadratic terms are introduced there is the potentiality of introducing important new concepts. For example, the "advantage of two bishops" could have been discovered this way, and it took chess players hundreds of years to formulate.

In more detail, let us label the pieces: WK, WQ, WKR, WQR, WQRP (White's Queen's Rook's Pawn), etc., and define x_{WQRP} as 1 if the WQRP is on the board, and $x_{WQRP} = 0$ otherwise, etc. Then (allowing only for material for simplicity), let us consider the quadratic expression

$$\xi = c_{WQ}x_{WQ} + \ldots + c_{BKRP}x_{BKRP} + c_{WQ,WKR}x_{WQ}x_{WKR} + \ldots \ .$$

By comparing the scores ξ of positions with outcomes of games, or with grand-master's evaluations of the positions, we could optimize the coefficients, using

multiple regression. We should discover that $c_{WKB,WQB} > 0$ whereupon we would define a new *primitive* variable $x_{WKB,WQB} = x_{WKB}x_{WQB}$ to be used in the next round of calculations.

Thus, once such a "quadratic concept" has been discovered it can be *transferred to the list of primitive concepts*. In this way one might then discover a concept involving three or more of the original primitive terms without ever using evaluation functions of degree higher than the second. If the number of primitive terms were say one hundred, there would be about 5000 quadratic terms and about 125,000 cubic terms, so it would presumably be impracticable to use cubic terms from the start. Although this idea of using quadratic evaluation functions would not always work it is worth following up because of its simplicity.

In the discussion, Michie pointed out that there is a strong analogy with the genetics of evolution. New structures of enormous complexity have evolved by means of mutations, occurring one at a time, but building up on each other. A series of mutations, each of slight effect in itself, can be found to be beneficial to fitness *in combination*. Genetic mechanisms tending to glue such combinations together, conceptually similar to the formation of quadratic terms, have been described.

Another discussant pointed out that John Holland has used this notion of quadratic terms as a method of concept formation (Holland, 1975).

REFERENCES

Good, I.J. (1959) Could a machine make probability judgments? *Computers and Automation* 8, 14-16 and 24-26.

Good, I.J. (1967) A five-year plan for automatic chess. *Machine Intelligence* 2 (eds. E. Dale and D. Michie; Oliver and Boyd), 89-118.

Holland, J.H. (1975) *Adaptation in Natural and Artificial Systems* (Ann Arbor: University of Michigan Press).

PROGRAMMING TOOLS FOR KNOWLEDGE-REPRESENTATION

13

Programming Language Design for the Representation of Knowledge*

J. M. Foster

Royal Radar Establishment
Malvern, U.K.

INTRODUCTION

In recent years there have been a number of programming language designs aimed at artificial intelligence research. In this paper I shall discuss some of the features of artificial intelligence work which have impelled the design of new languages, and what kind of designs have resulted. I shall take for granted properties that all languages should have, such as an appropriate conciseness, localisation of decisions and so forth; nor am I concerned with the highly important question of the system in which the language is embedded and the way it is used to produce programs and the influence of these on the language design.

It seems likely that features such as those discussed will appear in conventional programming languages in due course. The requirements of artificial intelligence are not special to that discipline, but are merely in a more advanced stage and more ambitious than in other areas. So we may certainly hope that satisfactory solutions will emerge and become more widespread through languages.

This is not a review paper for the existing languages, but it would be inappropriate not to mention the languages on which these abstractions are based. In particular, Planner (Hewitt, 1971), Conniver (McDermott and Sussman, 1972), Leap (Feldman and Rovner, 1969), QA4 (Rulifson, *et al.*, 1968), Qlisp (Reboh and Sacerdoti, 1973) and Abset (Elcock, *et al.*, 1971) have influenced these ideas. A review of Sail (Swinehart and Sproul, 1971), Planner, Conniver, Qlisp and Popler (Davies, 1973) is given by Bobrow and Raphael (Bobrow and Raphael, 1974).

Section 2 contains an outline of the needs for programming languages and the remaining sections discuss the various language features.

REQUIREMENTS OF ARTIFICIAL INTELLIGENCE WORK

A machine shows its knowledge by its behavior. As it goes through its party

*Copyright © Controller HMSO, London, 1975.

piece—light-switching, banana-fetching or block-stacking—we judge it on what it can do, not on what it could do with unlimited time and money. So a program's knowledge resides in the information it has and also in the strategy it uses to deploy this information. In designing languages for artificial intelligence work we must be sure that they contain adequate means for expressing not only appropriate data structures but also control structures.

Searching and models

Most artificial intelligence programs have to do some searching. Very often this searching has a structure: in order to show something we search for a number of subgoals.

The binding of names in the programming language to values during evaluation is organised in conventional programming languages on the basis of procedures. Entering a procedure with certain values as actual parameters binds the formal parameter names to these values and the body of the procedure is evaluated with these bindings (local names and blocks form a trivial extension). Calling another procedure forms more bindings and so at any particular time during the evaluation the value bound to a name can be found in the most recent binding or somewhere in the linear chain of bindings which ascends to the outermost level of the program. Recursive calls of procedures are deemed to generate new copies of the names and so do not affect this picture.

We can very easily apply this kind of control structure to searching a tree. Each node in the tree corresponds to a procedure which in turn calls procedures to process its immediate descendants, usually recursively. The computation at any time must depend only on the nodes which have been processed and those which are partly processed and a node once processed is finished with. Such an organisation of searching (depth-first) does occur quite frequently and where it can be used it is efficient from the language point of view. Such programs are also clear, comprehensible and easy to write. A typical example is the searching of minimax game trees. However, GPS (Newell and Simon, 1963) searched in this way, and because once having chosen a goal it was forced to continue with it and could not try another line of attack (by the very nature of depth-first search) it frequently became trapped. There are indeed many cases where the searching cannot be organised in this simple fashion; we may not know an appropriate order in which to scan (as in GPS) or the data may not be so simply hierarchic. Indeed if we choose an unsuitable way to scan an infinite tree we may find that we have to perform an infinite search before arriving at the desired goal. So although depth-first search corresponds to a common sub-goal organisation it is by no means sufficient.

Another common form of searching throws away the sub-goal organisation and works in terms of a set of 'states' which to the searching programs are uniform. A step of the search selects a subset of the states and as a result of some processing removes states and adds new ones to the working set of states. This step is repeated until some desired combination of states appears. Here the

states are on the same footing and are self-contained. Whereas in the tree search the bindings of the procedure calls gave a context for the processing of each node, in this algorithm the processing is a simple loop and carries no such context. Such information has to be carried around in the states themselves. Examples of this method of searching are also common, e.g., the early Logic Theorist (Newell, Shaw, and Simon, 1957), the Graph Traverser (Doran and Michie, 1966) and resolution-based predicate calculus theorem proving programs. However, this uniformity and the loss of the sub-goal structure in turn presents problems. We have to formulate the whole problem in a uniform way.

Consider one of the difficulties that arose in applying theorem proving techniques to real-life situations, the 'frame problem'. If we formulate, for example, the monkey and bananas problem in terms of axioms about the monkey-box-banana world and the possible actions in it, we find that we have to include such axioms as 'when the box is moved the bananas stay still' and 'when the monkey stands on the box the bananas stay still' and so on. We have to describe the total effect of any action, saying not only what it changes but everything that is unaltered. In a complex world this would lead to a very large number of axioms which makes finding proofs much more difficult, as well as increasing the size and complexity of the axiomatisation. Yet these extra axioms are logically necessary for the proofs we want to make. The difficulty arises because we have to put all the information into the uniform system of predicate calculus clauses.

We have, then, two extreme positions: one is convenient and carries contextual information but is too rigid as to order, the other is flexible about order but makes the problem difficult to represent and throws away structure (sub-goal structure) which the problem may be known to have.

For a long time these were the main searching methods used, in spite of hints from simulation languages. But clearly we would like to have the advantages of both systems, to have the organisational advantage of contextual information, and to get the freedom to dart about among the various lines of attack on the goal according to what seems at each moment most profitable.

Consider a block-stacking world as an example. We may wish to explore the effect of actions on the model, leave the exploration and start another one, abandon that and resume the first, undo some changes in it and their consequent effect and continue, and finally perhaps perform some plan in the real world which may itself be interrupted by some unexpected effect, leading to more planning, more actions and so on. To do this we need to be able to treat partly evaluated program with its context as data, and at least be able to park it and to reactivate it.

Furthermore, if as a result of some sub-exploration a general fact has been discovered it should be possible to publish this information to the other explorations in progress, lest it be repeatedly discovered. So the contexts must not be totally separated from each other.

Relational structures and matching

We must also consider what sort of information we want to keep in a context

(which is at least partly a model of the limited world we are exploring). If a value is completely defined by an initially determined set of properties, such as the length and direction of a line, and all we wish to do is to bind names to such values, then the mechanisms of conventional languages will suffice. A value can be a record structure (or a simple value such as an integer) and the selectors of this structure will give the values of its properties. This is an efficient way of representing such a situation and we will want to use it whenever we can. But there are many situations where extra information about things is produced during the course of operation. We may find or postulate that a group of objects stand in some relationship to each other, or that some objects are equivalent, or that a value has a property which is not of the totally defining nature. In these cases we want to manipulate information about objects, still in the sense of a context, which cannot be slotted into predetermined holes in a fixed data structure, which is the complete description of an object.

If we can store such relational information we must also be able to interrogate it, for example to ask (always in a context) 'is A near B?' or 'what points are near A' or in general 'what objects stand in this relation to these given objects?' or even 'do these objects stand in any relation to each other?'. This raises the question of how such queries are to be formulated.

If we take a pattern to be a relation in which some of the components are already chosen and others remain to be bound, we can formulate the above requests as 'find all the bindings in this pattern which yield relations which are present in this context.' Clearly for such a formulation of the query it is not necessary for relations to be present in some concrete data-structure sense, it is merely enough to provide an algorithm which will deliver sets of bindings.

Programs as objects

The designer of any language must consider how programs in it can be manipulated, for example, we may wish to compile them, to modify them, to prove assertions about them, to index them and so forth. Too much freedom to do anything to any objects makes manipulations difficult. This is an aspect in which conventional languages, though bad, are better than the artificial intelligence languages, because the appropriate constraints, which permit manipulations and also permit us to write the programs we want, have not been found yet.

We may consider a programming language in operation as consisting of primitive procedures and an interpreter. We make compound procedures by presenting structures to the interpreter which controls the binding of names to values and the precise sequence in which (eventually) the primitive procedures are applied. The interpreter specifies the control structure. The primitive procedures and the relations between them specify the data types: integers, reals, lists, stacks, etc. This distinction cannot be maintained if we permit the primitive procedures to manipulate partly evaluated program, since part of the responsi-

bility for control then devolves on them. Such procedures also make the task of proving things about (and understanding) programs very difficult. At present it does not seem to be clear how to design an interpreter which itself satisfies all the control requirements, but to do so must certainly be an aim.

CONTEXTS

Let us first consider simple binding, that is of identifiers to values, but in such a way that a context can be temporarily halted and later picked up. This is no different from the requirements of simulation languages. A more detailed discussion than this is given by (Bobrow and Wegbreit, 1973) who also propose a particular implementation.

We will suppose that associated with each activation of a procedure is a block of store which contains space for

1) The values to be bound to the local names
2) A pointer to the block belonging to the procedure which called it (access link)
3) A pointer to the block which is to be used for binding the immediately non-local names (binding link)
4) The place at which the body of the procedure is to be reactivated.

Then we may find the value bound to a particular name by first looking in the current block, then in the block pointed to by the binding link, then in the block pointed to by its binding link and so on. We may reactivate the procedure by starting it at the point specified, and we may return to the caller by reactivating it (possibly giving it a value as well). However, these blocks cannot in general all be allocated from the same stack. They can easily be allocated from a heap (a garbage collected area) or one can use the scheme of Bobrow and Wegbreit, which reduces to stacking in the case where this is possible.

Though we deal with relational structures in the next section we remark that the implementation must have the same properties as above, that is, the context must specify a particular relational structure, so that reactivating a context resets its particular relational structure. An efficient way of treating this for two place relations is suggested by Wegbreit (Wegbreit, 1973).

Note that removal of a particular relation between objects must remove it only in the particular context, not in surrounding contexts.

A further possibility we may require is that assignments to components of data structures be localised to the context in a similar way. This is much more difficult to implement unless control passes between contexts in more simple ways than we need.

RELATIONAL STRUCTURES AND PATTERN MATCHING

We shall consider briefly the association scheme used in Leap (Feldman and

Rovner, 1969). This system is an extension of Algol 60. It contains new types, some new operations and an elaborated **for** statement. The most important of the new types is a universe of 'items.' Items can be introduced

1) By declaration. The declaration is processed at compile time: the items are not produced when the declaration is encountered dynamically.
2) By an explicit dynamic operation.
3) By creating an association item from three items, commonly written attribute. object≡value

An algebraic value (integer, etc.) can be tied to an item and obtained by an operation. Items which were explicitly created can be explicitly erased, though declared items cannot be erased. Otherwise items last indefinitely.

Other new types are the 'set' of items, 'itemvars' which are variables of which the values are items and 'locals' which enter into the **for** statements in a peculiar way. The simplest of the new **for** constructions operates directly on a set. If s is declared to be a set and x is declared to be a local then

foreach x **in** s **do** statement involving x

will execute the statement with x substituted in turn by each of the items in s. But much more general looping statements are available. For example, if a and r are items and x is a local, then

foreach a.x ≡ v **do** statement involving x

will execute the statement with x substituted in turn by the items i for which the triple a.i ≡ v occurs in the whole collection of triples at this particular moment. And if a is an item and x, y and z are locals, then

foreach a.x ≡ y **and** a.y ≡ z **do** statement involving x,y,z

will execute the statement with items i, j, k substituted for x, y, z in all the ways such that both triples a.i ≡ j and a.j ≡ k occur in the whole collection of triples. These are clearly very powerful constructions, though there is some awkwardness in the way the control sections are given (this has been changed in Sail, which derives from Leap in these respects).

It is possible to interrogate the data-base of triples in seven ways corresponding (where x, y are locals and a, b, c are items) to

a.b ≡ c, a.b ≡ x, a.x ≡ c, x.b ≡ c,
a.x ≡ y, x.b ≡ y, x.y ≡ c

The eighth possibility (x.y ≡ z), which would give the whole set of associations,

seems not to be available.

It is worth consulting the paper for the ingenious way in which the triples are stored and the requests serviced. However, it is this which constrains the system to use only triples (though anything can in principle be represented in this way) and to have a global set of triples with explicit addition and deletion, i.e., no contexts.

In spite of the constraints, we can see the elements of working with relational structures.

1) A relation is a sub-set of a product of n sets. In Leap there is only one relation (at one time), which is a sub-set of Item x Item x Item. Though this is general in the sense that we can represent any relation thus, together with the association of values with items, we would prefer to have many relations and no restrictions on the number or type of sets involved in the product. For example, a sub-set of Item x Integer should be a perfectly satisfactory relation.
2) As well as the various sets of values naturally available in any language we need to be able to introduce new kinds of items which serve only as markers in various relations. Leap permits one kind of such items, but we may want more.
3) We need to interrogate the relations. The simplest way is to ask whether some specific n-tuple is present in the relation. But we want, more generally, to ask for the intersection of the relation with some other subset of the same product space. For example, where in Leap we said

   foreach a.x ≡ y **do** statement

   we were finding the intersection of the relation (the triples) with the subset of Item x Item x Item determined by making all substitutions of items for x in the form a.x ≡ y. We therefore need various ways of specifying sub-sets and forming the intersection with a given relation.
4) Since the result of such a query is a set, we must be able to manipulate sets. There must be suitable set operations in the language. Of course, there is no implication in this that any of these sets are necessarily represented in extenso. We need at the least a way of doing something to each member of a set, and this is in effect what Leap provides.
5) We must be able to make and modify relations, for example, by adding or deleting specified n-tuples or subsets.

Of the questions raised by these needs, the most interesting is how we should specify the subsets which we want to intersect with a given relation. Four

methods immediately arise.

1) By explicitly enumerating a set of n-tuples. Requests deriving from such a set can be serviced by enquiring whether a particular n-tuple is present for each member of the set.
2) By giving a predicate on n-tuples. Requests cannot, in general, be serviced very efficiently; it will be necessary to run through the relation and apply the predicate to pick out the desired n-tuples.
3) By giving a pattern. Here we specify certain members of the n-tuple. If a and b are fixed members of A and B, then we could ask for a subset of A x B x C x D such that the elements are of the form, (a, b, x, y) for all possible substitutions for x, y from C, D. We could call (a, b, x, y) a pattern for the subset. Clearly we are really interested only in the subset of C x D. This is illustrated in some of the Leap **foreach** statements. Requests can be serviced fairly efficiently if the relation is stored in such a way that it can be suitably indexed.
4) By combining subset specifications using set operations such as intersection, union, complement, product. This again is illustrated by the Leap **foreach** statements.

Note that the subset of A x A of pairs of equal elements can be specified by a predicate, but we might like to specify it by means of a slight generalisation of patterns as (x, x), a feature again available in Leap.

We can elaborate the scheme slightly by distinguishing one of the spaces and writing an element of a relation, R, in the form R(a,b,c,d) = e, where the equals sign carries no implication of single-valuedness for R. We could then write patterns like T(x,S(a,x),b), and we can think of $R^{-1}(e)$ as another way of defining a subset of A x B x C x D. Of course, this can be written with a pattern, but it raises the possibility of defining subsets as inverse operations.

The rather awkward way in which the locals are bound in Leap prompts a discussion of the binding of the substitutable identifiers in a pattern. We have spoken in terms of intersecting a pattern-defined subset with a given relation to obtain a sub-set, to each member of which we then apply some operation. Since we are not interested in the members of the tuple which were fixed in the pattern, and since we have always some operation to apply to the substituted values, we could choose (in the case of patterns) to group the operations differently. That is, we would group the pattern and the operations together, e.g.,

bound identifers	pattern	action
λx	(x,x)	print(x)

into one procedure-like object, and apply it to the relation. This would be interpreted as applying the action to those substitutions for x which gave a tuple

which was in the relation. This way of binding certainly seems clear, and such procedure-like objects can be used in other ways, for example, they could be themselves stored or passed as parameters.

Clearly, from what we have said, the relation and the request are in symmetric positions, but various reasons impel one to use different kinds of subsetting for the two situations. In the request we commonly use a pattern, perhaps qualified by predicates because this enables us to index a relation given *in extenso* and so obtain a relatively fast look-up. In the relation we commonly use an explicitly given set of tuples. This is partly because it can be indexed. It is also because of the nature of the kind of models we wish to generate. Neighbouring contexts commonly differ by having a few explicit additions and deletions made to their relations. Furthermore, this helps in moving from one context to another–the changes can hide the relevant part of the previous relations without the necessity of reorganising them.

But we may also want to give an algorithm for the relation. Here we can find another use for the procedure-like pattern-action objects. We can use these to say that a relation lies in the subset given by the pattern and that the action will generate members. Then if we make a pattern request of such a relation we can combine the two patterns together (in the sense of the unification algorithm of resolution theorem proving) and use the resulting pattern to control the action. This then permits us to index these relations also. Such a mechanism is given by the 'if-needed' procedures of Conniver. To combine two patterns together we seek substitutions in each, which will make them into the same pattern. If such substitutions do not exist then the set-intersection is empty. If they do exist then we can always find a most general one, in the sense that all other such substitutions can be obtained by making substitutions in the most general one.

It must be made clear that, though we speak of finding a subset and applying operations to the members of it, we are not implying that the whole of the subset is developed and the operations then applied. We must certainly be able to generate the subset piecemeal, meanwhile applying the operation to such elements as are available. We return to this point in the next section on sequencing.

Planner and Conniver use strings of words as elements of one universal relation (for a particular context) rather than n-tuples of values as the elements of several relations. This, because string concatenation is associative, allows them to match not only individual words but also sequences of words of unspecified length. They also allow the order in which individual pieces of a pattern are matched to play a part, as Comit and Snobol do for their string matching operations. Nevertheless, they approximately follow the use of relations and patterns which we have described.

CONTROL AND SEQUENCING

The **foreach** statement of Leap, though it is controlled in a novel way, is conventional in the matter of sequencing–it is a simple loop. Indeed Leap, being

based on Algol 60, inherits the sequencing structure of that language. Other languages have more novel sequencing.

One such novelty is the use of *demons* (a Qlisp word, though the idea is present elsewhere). These are procedures which are automatically invoked when something specified happens, for example, if a certain value is changed, some condition becomes true, or some relation is altered in a particular way.

We have seen that we need operations on a relation which add new tuples to it or remove ones which are present. We could attach to the relation a 'demon' which when a new element is added lying in some subset wakes up and performs an action. If we specify the subset in question with a pattern, we have another use for the procedure like objects formed from a pattern and an action. Such a value could be added as a demon to a relation. Similarly we could arrange to execute an action if a suitable tuple is removed. These demons are particularly useful when the relation is part of a model of a real-life situation. For small changes to such a model often propagate to produce a number of resultant changes in a restricted way which can be conveniently expressed in terms of demons. Difficulty in treating this in predicate calculus was previously mentioned as "the frame problem." We have gained because we have a relation which can be altered to represent the current state of the world. Qlisp permits teams of demons to be specified for every data storage and retrieval operation.

Deductive mechanisms

We shall consider the deductive mechanisms available in Planner and the sequencing that they require.

In Planner we can assert facts, such as

< ASSERT (HUMAN TURING) >

Though Planner deals with strings of words rather than with n-tuples, in many examples and in particular those which follow the strings are used as if they were n-tuples. So the above assertion we shall interpret as Turing is a human. We can also assert theorems, such as

< ASSERT < DEFINE THEOREM 1
<CONSEQUENT(Y) (FALLIBLE ?Y)
<GOAL (HUMAN ?Y)>>>>

This is to be read as meaning that if we want to show that something, y, is fallible we should try to show that y is human. Note that we are using patterns here. If we now evaluate

< GOAL (FALLIBLE TURING) >

Planner will first look to see whether

(FALLIBLE TURING)

has been directly asserted, and if it has not it will look to see whether any theorems (consequent theorems) have been asserted for which the pattern, in this case (FALLIBLE ?Y), can be made to match the request (FALLIBLE TURING). Since substituting Y by TURING produces a match, Planner evaluates

< GOAL (HUMAN TURING) >

which can be directly answered from the explicit assertions. Planner thus concludes that the original goal could be satisfied.

We see immediately one of the important features, the GOAL statement which tries to prove something, and also the automatic way in which it brings into play the 'consequent theorem' which, however, was asserted separately. Of course, these theorems can be activated to any depth and in any degree of complexity.

The GOAL statement was here a question about a particular word, TURING. It is also possible to ask questions which in logic would be formulated with existential quantifiers, such as "is there a y such that y is fallible?". This is written in Planner as

< THPROG (X) <GOAL (FALLIBLE ?X) >>

THPROG acts like an existential quantifier. To evaluate it Planner looks for something explicitly asserted which matches (FALLIBLE ?X). Since there is nothing, it looks for a consequent theorem with a pattern of the same form and unifies the patterns, in the sense of section 4, that is X and Y are identified. Then the goal (HUMAN ?X) is set up, still under the control of the THPROG. Since this can be matched with (HUMAN TURING), X and Y are bound to TURING and the whole THPROG succeeds.

Notice that there might have been assertions in the data base which encouraged Planner to try a line of attack which ultimately proved futile, while there were other satisfactory assertions present. Since Planner has no knowledge of an appropriate order for the particular problem in hand, it must exhaustively search the possible ways of finding the goal until it finds one which succeeds or until there are no more. Hence a failure must propagate back to the immediately senior place which caused a decision to be made, such as THPROG choosing a particular assertion to work on, and the work done since then must be abandoned (including undoing side-effects). Then the next choice is made and so on. Planner made these choices automatically, the way that they are done is built into THPROG and the other Planner statements, though the sequencing can be influenced by giving advice in the program.

Of course, we ought to beware of using complex and powerful constructions like these when simple and more efficient ones are available. A **foreach** statement, as a way of doing these things for which it is appropriate, is both more understandable and more efficient. One should only use these methods if **for** statements are not satisfactory.

The automatic deduction mechanisms built into Planner are both a strength and a weakness. If the operations it chooses to do form a satisfactory strategy for the problem in hand, then they can be expressed quite clearly. But if we need to use other strategies, or to skip about between various approaches, then things are not so easy. Because the methods are so powerful it can be very difficult to circumvent them, and indeed the very need for circumvention points to a degree of unsatisfactoriness. For example, the strict back-tracking makes it very difficult to communicate between tracks. So if a sub-exploration finds some general fact which it would like to publish, but fails in its own goal though the general fact was true, nevertheless the back-tracking mechanism will prevent the fact escaping. Another example of difficulty lies in ensuring that all the failures are understood. Failure to achieve a goal controls the sequencing, but if a trial fails for some reason unforseen by the programmer then the sequencing will be affected. Frequently this will cause other trials to be made which should not have been made, and the whole search may well succeed, leaving the only effect of the programmers error as an enormous slowing down of the running speed of his program, a not very satisfactory state of affairs.

Conniver's sequencing attempts to retain the good features of Planner but allows for more explicit control by the program writer. There is a data type called a 'frame' which carries bindings of names to values and a data base context. It also contains a control stack. Pointers to such objects can be manipulated by the programmer. There are also 'tags', which consist of a frame and a 'program counter' or place in a body. Tags can be generated explicitly by the function TAG which takes a label as a parameter and produces a tag with the current frame and program counter corresponding to the label. The function GO applied to a tag will reactivate that frame and start obeying the program at the program counter. All the Conniver control structure is built up on these ideas as basis.

Clearly this allows for very free composition of programs, but it imposes very little discipline on programmers (deliberately).

DATA TYPES

The data types needed in artificial intelligence do not really differ from those needed in general. The relations discussed above, considered as a data type, are exactly what is required in many normal programming situations which we now get over by various ad hoc devices. Exactly the same is true of the use of sets and lists. Bags, introduced in QA4, could also well become more general. Bags are collections of elements in which order does not count but the multiplicity of occurrence of an element is significant. So a bag of integers is a suitable argument for '+' which is associative and commutative, just as a set of booleans is a suitable argument for 'and' which is also idempotent.

Lists, sets and bags are not types, really, but type constructors, in the sense that we might have a list of integers, or a set of elements from the universe, or a bag of lists of booleans.

CONCLUSIONS

At present the control and binding ideas which we seem to need have not been designed into a system which is both sufficiently flexible to be used and also makes programs susceptible to analysis. Either they are too rigid, as strict back-tracking is, or too permissive like Conniver, or they force all these difficulties out of the programming and into the data, like resolution theorem proving.

I believe that we should be aiming for languages with the following features.

1) They should be based on a single interpreter or compiler (as Qlisp is and Conniver is not), so that the basic control structures we need are already available and we can write our artificial intelligence systems by writing packs of procedures.
2) That we should prefer the simpler control structures (like **for** statements) to elaborate ones, and design the language to promote them. But we should not be forced to generate lists *in extenso* as a result of this preference.
3) That we should try to categorize the more complex control structures we need, since, though we may have adequate primitive ideas, we do not have much understanding of their patterns of use.
4) That the languages should be carefully designed to permit transformations on programs and analysis of programs. We must be particularly careful here about binding, since prominent languages have failed in this respect.
5) That we should continue to try to build in relational structures and consider them in the light of garbage collection.
6) That we should study the use of patterns and their manipulation, for example in unification and indexing, to obtain more use from them.

REFERENCES

Bobrow, D.G., and Raphael, B. (1974) New programming languages for artificial intelligence research. *ACM Computing Surveys,* 6, 3, 155-174.

Bobrow, D.G., and Wegbreit, B. (1973) A model and a stack implementation of multiple environments. *CACM,* 16, 10, 591-603.

Davies, J.M. (1973) Popler 1.5 Reference Manual. *TPD Department Memo, No. 1,* University of Ediburgh.

Doran, J.E. and Michie, D. (1966) Experiments with the Graph Traverser program. *Proc. Roy. Soc. A,* 294, 235-259.

Elcock, E.W., Foster, J.M., Gray, P.M.D., McGregor, J.J. and Murray, A.M. (1971) Abset, a programming language based on sets: motivation and examples. *Machine Intelligence* 6 (eds. Melzer, B. and Michie, D.) Edinburgh University Press, Edinburgh.

Feldman, J.A., and Rovner, P.D. (1969) An Algol-based associative language. *CACM,* 12, 8, 439-449.

Hewitt, C. (1971) Procedural embedding of knowledge in Planner. *Proc. IJCAI2,* London.

McDermott, D.V., and Sussman, G.J. (1972) The Conniver reference manual. *AI Memo no. 259.* M.I.T. Project MAC.

Newell, A. and Simon, H. (1963) GPS, a program that simulates human thought. In *Com-*

puters and Thought. (eds. Feigenbaum, E. and Feldman, J.).
Newell, A., Shaw, J.C., and Simon, H. (1957) Empirical explorations with the Logic Theory machine. *Proc. WJCC*, 218-259.
Reboh, R. and Sacerdoti, E. (1973) A preliminary Qlisp manual. *SRI AI Center Tech. Note 81*.
Rulifson, J.F. et al. (1968) QA4, a language for writing problem-solving programs. *Proc IFIP Congress*.
Swinehart, D. and Sproul, B. (1971) Sail. *Stanford AI project Op. Note 57.2*, Stanford University.
Wegbreit, B. (1975) Retrieval from context trees. *IPL*, 3, 4, 119-120.

14

Some Observations on Conceptual Programming

Erik Sandewall

Informatics Laboratory
Linkoeping University, Sweden

One fairly large LISP program is analyzed carefully with respect to data structures, program structure, and implications on programming methodology. The program does a fairly conventional data-base management job, and was originally written for its practical purpose, rather than as a methodology experiment. The analysis is performed in the framework of conceptual programming, which says, e.g., that information about program structure and about data structures (for example the information commonly found in declarations) should be stored in the user-accessible data base of the programming system, and be usable in multiple ways. The long-range goal is to enable the programming system to "understand" the program, and to communicate with the programmer on his natural conceptual level. The analysis of the sample program results in suggestions for a number of programming methods, for example to use a second-order declaration structure as the top-level structure of the program, and then associate pieces of code with entities in the declarations, or structures formed from them. Another suggestion is to distinguish between an "execution" model and an "initialization" model of the system. The former is basically a combined data-flow and procedure-call structure; the latter is an idealized "program" for the conduct of an interactive session, plus a collection of inserts into or updates of that program.

INTRODUCTORY REMARKS

The work reported here consisted in a detailed analysis of an existing program of non-trivial size, resulting in a number of suggestions for program structure and program development methods. Since they are only based on observations in one program, the suggestions have the status of hypotheses rather than proven facts, and additional verification or falsification is desirable (and intended).

A study of this kind could probably not have been performed without a certain "ideological" background. One cannot observe merely by looking; one needs a conceptual framework in order to select what to look for, and to interpret what one sees. In the present study, my framework was the long-range ideal of computer-supported program development. I envisage a situation, at least in the limit, where the primary representation of the computer program is

as a structure in the computer. The user would only in exceptional cases see the whole program, i.e., a dump of the whole structure. Usually he would build it up piecewise, and inspect it piecewise by requesting that specific sections or projections of the structure be printed out. The programming system should "understand" the program structure well enough that it can communicate with the user about the program in a mature way, using the same conceptual level (although not necessarily the language) that two programmers would use between themselves.

This ideal is sufficiently vague and remote that it should be considered as a direction of movement, rather than a specific goal that one will reach and prove that he has reached. I adopted it as an ideal for two reasons:

- It is the only possible way to make the computer support the programming activity. There is a clear need for very-high-level programming systems, which help the programmer (or a team of programmers) keep track of large and complex programs. Winograd (Winograd, 1974) has successfully argued this point. At the same time, since a program is a richly connected structure of goals, decisions, conventions, solutions to sub-problems, and compromises, one cannot expect a programming system to understand one part of that structure if it does not have access to the other part. The system must have a full understanding of the program, or it will not have any useful understanding of it at all. Also, it seems to me that the only possible way of entering that structure into the computer is either to let the computer generate it itself (i.e., fully autonomous automatic programming) or to let the programming system participate in the program development under the guidance of the programmer. The first alternative is very remote, and the latter alternative is somewhat more realistic. The second alternative might also be a good subgoal for the first one, although that is not part of my motivation.
- The second reason is that, by developing an appropriate structure for the representation of a program in the above mentioned type of programming system, one is forced to make a precise analysis of structures which would otherwise just be intuitively understood. In designing such a "conceptual program structure," one is encouraged to ignore all the many trivial aspects of textual representations and syntactic sugaring, and focus on what are the essential structures in programs.

The term *conceptual programming* is chosen for the proposed programming style, where the program is represented as a "conceptual structure" in the programming system with which the programmer interacts for program development. This "ideal" programming style is not taken entirely out of the blue. First, the LISP programming language (and its cousin, the Vienna definition language) may be viewed as first steps towards that ideal. More important, several widespread although previously undocumented practices of LISP programming take additional steps in the same direction. Similar development can and does take

place in the framework of other languages, although more slowly and with much greater difficulty.

The conceptual programming ideal has been used as the framework for the program analysis that is reported here. Conversely, the present case analysis resulted in concrete suggestions for program organization and programming methodology, which make the conceptual programming idea(l) more specific in at least some ways.The sample program, written by Dave McDonald at MIT, uses those current programming practices which I view as steps in the direction of conceptual programming. The study resulted in some results regarding the extent and limitations of present practices, but more important, it generated a number of concrete suggestions about how these methods could be extended and improved.

This work was done in the context of the programming language LISP, a language that is used intensively among some groups of researchers, but rejected or shrugged off by many others. I believe that this is largely because all available textbooks on LISP are bad, and describe the language from an uninteresting and irrelevant point of view. Part I tries to make up for that by presenting what in my view are the significant properties of modern LISP system, for the benefit of readers who are not immersed in the LISP culture. Part II describes and discusses the inspiration of this work, that is it describes the current programming practices, and extrapolates to the long-range ideal. Part III reports on the detailed study of McDonald's program, and is the main section of the paper. Part IV is short, and attempts to summarize the results as a list of specific findings.

PART I

LISP AS A BASIS FOR CONCEPTUAL PROGRAMMING

LISP systems have certain properties which make them suitable as an environment for conceptual programming, which explains why the conceptual-programming trend has developed in the LISP-using community. Other languages such as SNOBOL or APL have the same properties and could presumably also be used for conceptual programming. Languages such as PL/I, Pascal, or Simula 67 lack the combination of those properties, and could not be used except after non-trivial modification. The purpose of the present section is to describe those properties which are essential for conceptual programming.

I think of LISP as determined by three basic design decisions. The first two are:

(a) for ease of debugging, the system shall be *incremental*, meaning that the programming system performs a read-evaluate-print loop, where in each cycle the user enters an expression, has it evaluated, and sees the result. The expressions may serve to define a procedure, store something in the data base, evaluate an expression in order to test a procedure, or edit a procedure or the data base. The user communi-

cates all the time with one single programming system, and does not have to switch between "edit", "compile", and "execute" modes.

(b) because of the intended applications, the language shall contain facilities for handling *data structures* and maintaining a *data base*.

Neither of these criteria is unique: APL and many implementations of BASIC satisfy the first criterium, and PL/1, Algol 68, Pascal, etc. satisfy the second criterium. However, *the combination of these two purposes is not trivial* to achieve. The reason is that the read-evaluate-print loop assumes that one can type in arguments of procedures to the programming system, and obtain their values typed out on the console. In order to also satisfy the second requirement input and output of data structures must then be defined–which it is not, in conventional programming languages.

In order to account for input and output, LISP encourages a different method of data structuring than record-oriented languages. Consider the traditional example of family relationships: suppose one wants to design a data base that maintains information about persons, and in particular, information about parent-child and brother-sister relationships. In record-oriented languages, the following structure is natural: each person is represented as a record, with pointers to other records, for example a "father" pointer and a "mother" pointer. Also, the circumstance that one person may have several children is handled by letting each person point to its "oldest child", and also to let each person (namely each child) point to its "next younger sibling". This is illustrated in Figure 1.

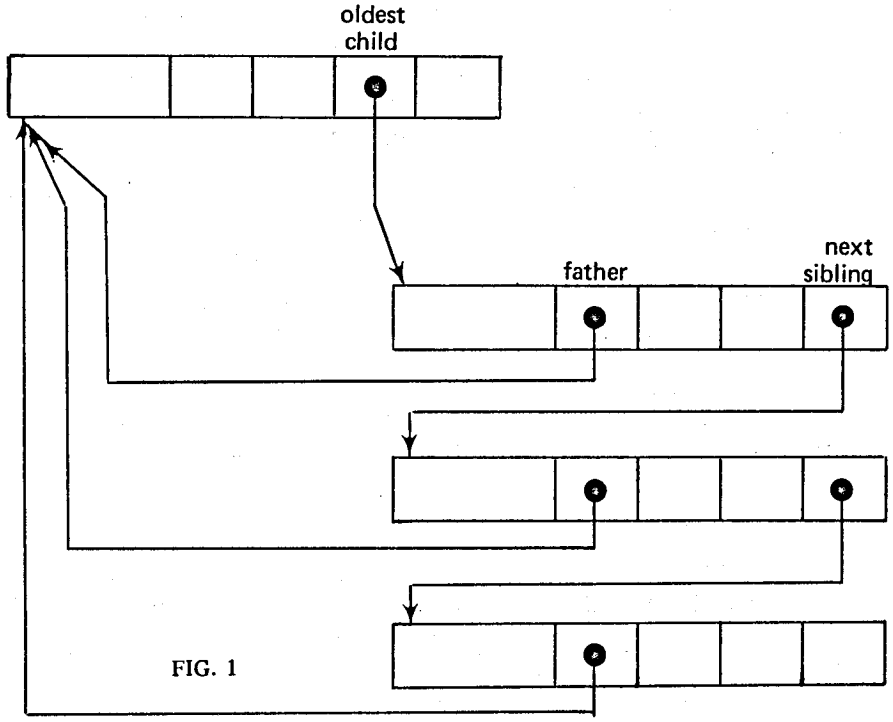


FIG. 1

In LISP, one is encouraged to use a built-in data type called an *atom*, i.e., a special kind of record which stands in a one-to-one relation with a character string. Thus each time the standard *read* routine encounters that character string, the same record is retrieved. In the present example, if JOHN's children are called BOB, DICK and MARY, and ignoring the problem that several persons may have the same name, one would have one atom for each of these names. Furthermore, the system contains two primitive operations, which we shall here call *get* and *put*, and which serve the following purposes: *put* is called with three arguments, and stores a property assignment in the data base, for example

put(BOB,FATHER,JOHN)

with the obvious intended meaning (John is Bob's father). Similarly, the function *get* retrieves a property from the data base, for example

get(BOB,FATHER)

which should return the value JOHN (i.e., the atom = the record which stands in a one-to-one relationship with the character string "JOHN").

So far there is no conceptual difference from the structure in Figure 1. The difference comes when handling the set of children of a person, where the LISP-oriented structure is to form the sequence of the children,

<BOB, DICK, MARY>

and to assign that sequence as a property:

put(JOHN,CHILDREN,<BOB,DICK,MARY>)

This sequence has traditionally been implemented using binary pairs of pointers, as shown in Figure 2. (More storage-efficient representations are presently being developed.) Thus the conventional record structure encourages one to represent sets and sequences of objects by *threading through* them, giving the structure of shark's teeth on a necklace, whereas the LISP-oriented structure looks like a comb with pointers *down* to the objects involved.

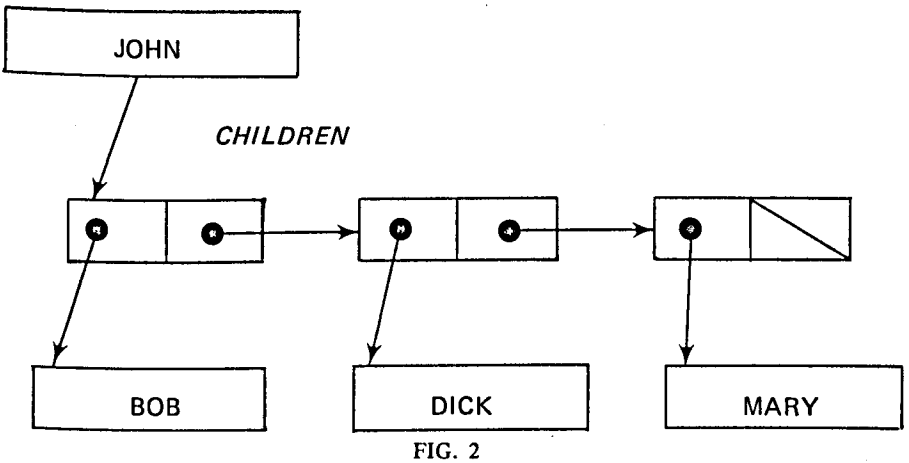


FIG. 2

It is not my purpose here to argue for one or the other of these representation models; they both have their merits and demerits. Let me point out, however, that it is not trivial to use the LISP data structure in conventional record-oriented languages. The structure of pairs of pointers can of course easily be implemented, and one can also write a program which converts a character-string to a corresponding atom, or uniquized record, by going through a symbol-table. However, that only accounts for atoms that appear in input data to the system, but not for atoms that appear as constants in the program itself. In many cases it is important to be able to use the same atom in the program as a constant, and in input data.†

The third design consideration for LISP is that there should be a standard convention for *representing programs as data structures in the language*, in order to facilitate generation and manipulation of programs.

The appropriate representation for a program is of course that of a tree. Thus the expression a + b x c + d is viewed as the tree in Figure 3:

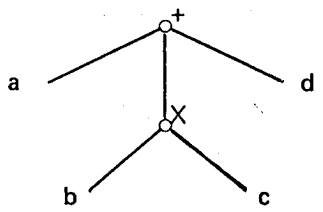


FIG. 3

which is encoded as the following data structure in Figure 4.

The boxes in Figure 4 that contain character strings are realized as atoms in the executing programming system, and in general, all atomic components of a program (procedure names, entities in declarations, variables, etc.) are represented as atoms. The atoms PLUS and TIMES stand of course for + and x.

For purposes of program analysis and generation, the data structure representation of the program is more convenient than the conventional representation as a character string, since the components of the programs (identifiers, and sub-expressions on different levels) have already been extracted, and are available as entities. The internal data-structure representation of the program is similar to an intermediate representation in a conventional compiler.

This program representation goes well with the first two design decisions. Since procedures are defined and edited by evaluating expressions in the lan-

†The situation is analogous to the handling of numbers in algebraic programming languages. A Fortran system allows input data to contain characters strings such as "4.65", and converts such a string to the internal representation in the computer, usually as the floating-point contents of a cell. It also allows such expressions to appear in the program, and if so, internizes them at compile time and saves them in such a fashion that they are accessible when the program is executed. An analogous mechanism for atoms is necessary in order to incorporate them into a compilation-oriented programming system.

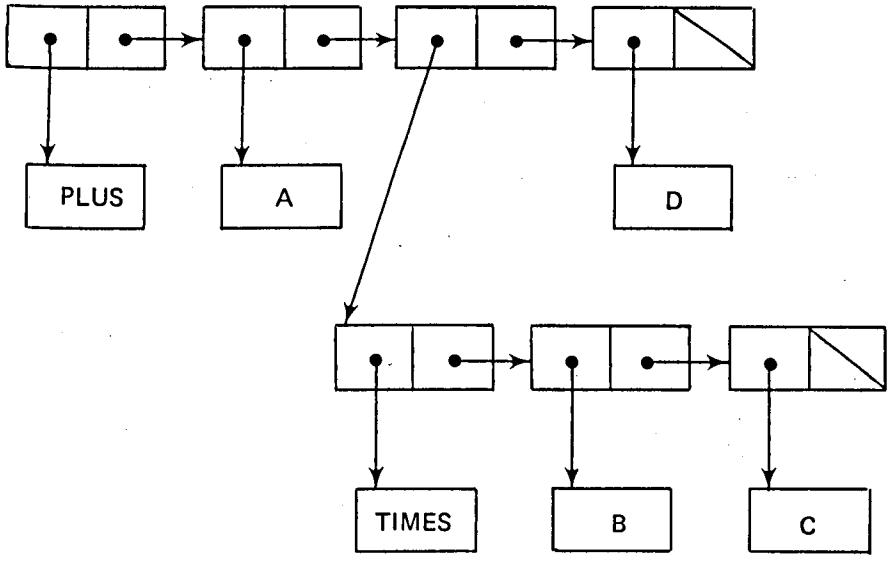


FIG. 4

guage, it follows that these operations do not *have to* be performed on the top level of interaction with the system; they may also be performed during the execution of a program. Experienced LISP programmers very often use that possibility for low-key program generation. Also, since all entities in the program are atoms, it becomes possible to associate information with program entities, for example to store descriptions of the data structure (the declarations in the program) in the data base. It becomes possible to write programs that inspect their own declarations, and each user can store arbitarary information (for example, for documentation purposes) with the declarative information.

The question of input of programs in LISP is often misunderstood, and shall therefore be discussed here. The data-structure representation of the program is the preferred internal representation, meaning that the interpreter and the compiler are defined to operate on it. There is, however, no commitment as to how the structure is to be entered. One possibility is to have an Algol-like language which is translated to that structure, and which may then have declarations, infix operators, for statements, and so forth. Translators for several such input languages have been developed. However, since input and output are defined for arbitrary data structures, one may also use that facility for entering and printing out programs. Using standard data-structure I/O, the above expression would be represented on paper as

<PLUS A <TIMES B C> D>

Many users prefer that notation, since it provides a more direct contact with the internal representation of the program, but the question of external program

notation is a matter of personal taste, not basic system design.

The two data types of atoms and binary pairs of pointers, are basic for the language, but they are not the only allowed types. The two basic types have a special status since the internal representation of programs uses them, and since input/output for them is predefined, but modern LISP systems such as INTERLISP (Teitelman, 1974) and LISP derivatives such as EPL (Wegbreit, 1972) contain facilities which enable the programmer to use other data types as well, and also to define rules for input, output, and evaluation of his own data types.

The basic operations *get* and *put* that were described above can easily be generalized. In essence, *put* associates an arbitrary expression with a pair of atoms, for example associates JOHN with the bituple <BOB, FATHER>. It is trivial to write a more general function *put** which associates expressions with arbitrary nested tuples, and not just with pairs of atoms. Modern LISP systems contain a number of such functions, implemented using either a tree or a hashing technique (or a combination of those) for storage of the associations.

The *put* operation first appears to be analogous to the assignment of terms in a record, when a conventional record structure is used. There are however several important differences. First, both the first and the second argument of *put* are arbitrary atoms, and the program may variablize with respect to either of them, or both. In a conventional language, if one has an assignment statement of the form

```
x.father := readstring();
```

then x is a variable (and may be a whole expression), but *father* is a constant. In the corresponding situation using the *put* function, one may have a variable that happens to be bound to *father*, or an expression that evaluates to it, or a loop of the type

```
for p in list-of-property-names do
    begin
    print(p);
    x.p := readvalue()
    end
```

Another difference is that the *put* operation is dynamic, i.e., the basic programming system does not maintain declarations that specify which properties are allowed for which objects. This is significant in an incremental environment. Suppose one is interacting with the programming system, and he has loaded a large program, and a data base which is being used when testing the program. He now decides to add one more property to a type of objects. With a LISP-style *put* function, he just goes ahead and stores those properties. If instead the properties had been frozen by declarations, he would have to:

- dump the data base (since the data base has probably been input interactively and incrementally during the testing of the program);

- edit the program;
- recompile the program;
- load the data base.

The same advantage is evident if another program performs the change of structure.

Thus the basic programming system does not maintain or use declarations, but at the same time, the properties of the system *encourage* the development of higher-level systems that use declarations. In the above example with family relationships, one might store in the data base

```
put(JOHN,TYPE,PERSON)
put(PERSON,PROPERTIES, <FATHER,MOTHER,CHILDREN>)
```

indicating that JOHN has type PERSON, and which properties are expected for that type. One may also store

```
put(FATHER,STRUCTURE,PERSON)
put(CHILDREN,STRUCTURE, <SEQUENCE PERSON>)
```

to specify the desired structures of such properties. Even in these very simple examples, one makes use of the fact that program entitites, such as type names and property names, can also be used as data. The fact that programs are represented as data structures facilitate the task of writing programs that check a program's consistency with its declarations. (Nordström, forthcoming, is doing that with Simula 67 as the input language).

More fancy declaration systems are possible and worthwhile, such as declarations which specify how other declarations are stored. A major theme of the present paper is how such declarative structures should be developed.

PART II

CONDENSED IDEAS ABOUT CONCEPTUAL PROGRAMMING

The ideas of coneptual programming are very much "in the air" in the artificial intelligence community. This section represents an attempt to "condense" some of those ideas, in order to set the stage for the report on actual work in the next section. The methods and ideas that are described here largely represent my own experience from working with programs such as PCDB (Sandewall, 1971,1973) and REDFUN (Beckman, *et al.*, 1975), but that experience closely parallels the ideas and experience of many others. It is hoped that many readers will experience a sense of *déjà vue* when reading this section.

Data-driven programs

A classical model of a program is that it is a collection of procedures which call each other. Each procedure has a name, and another procedure can call it by explicitly mentioning its name. The calling structure is statically available, so for

example it is possible to write programs which take a set of procedures and produce a graphical representation of the calling structure.

In data-driven programs, on the other hand, the procedure calls are indirect via the data base. One ("executive") procedure or program accepts input data, either from user input or as arguments, retrieves procedures which have been associated in the data base with data items that appeared in input, and executes those procedures. This indirect calling structure is illustrated in Figure 5.

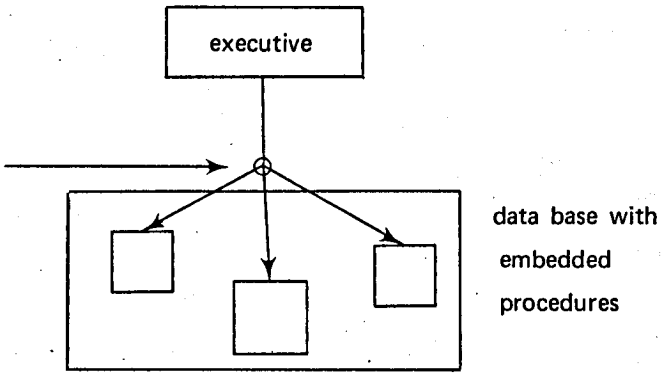


FIG. 5

There is an abundance of examples of this technique. The PCDB system (Sandewall, 1971, 1973), maintains a data base of assertions in predicate calculus, and assumes that each relation symbol is associated with a storage procedure (for storing the relationship in the data base), a retrieval procedure (for looking it up), a search procedure (which uses deduction to look it up), procedures for answering open questions, etc. In order to assert a relationship such as

COMPONENTS(finger,hand,5)

to the system, one calls a general-purpose procedure *store* with the arguments

store(COMPONENTS, FINGER, HAND, 5)

where store(r,x,y . . .) is defined to look up get(r,STOREFN) and execute it with (x,y . . .) as its arument list. Thus *store* makes an indirect call, or *dispatches* to procedures associated with relation names.

Programs that operate on LISP programs provide several examples of data-driven-ness. The REDFUN program (Beckman, *et al.*, 1975) performs partial evaluation and other simplification of LISP procedure definitions. It allows that procedure names in the program that is to be simplified, may be associated with specialized procedures which know how to simplify expressions with that procedure as its leading operand. Thus for a trivial example, the simplification procedure for PLUS would embed the knowledge that a + 0 = a. Prettyprinting programs (i.e., programs which produce nicely indented presentations of programs)

such as the one in MACLISP (Moon, 1974) are data-driven with respect to procedure names in a similar fashion. Risch (Risch, 1975) discusses a number of program-manipulating programs that dispatch on procedure names, and proposes a way of systematizing their conventions so that they can dispatch to the same set of procedures.

In the continued discussion we shall repeatedly use the same example, namely a program system written by Dave McDonald at the M.I.T. Artificial Intelligence Laboratory. The program maintains a data base of document descriptions, such as author, title, year of publication, and so on for several types of documents (books, articles in journals, internal research memoranda, etc.). The major purpose of the system is to take a list of document identifiers, and to print out the list of the author, title, etc. of those documents, in a format which is suitable for the list of references at the end of an intended new paper. In particular, the printout program can be instructed to conform with any of the idiosyncratic sets of rules that different journals impose on authors (first name of author before or after last name; names of journals must or must not be abbreviated; and so on). The system is called the bibliography system.

The system maintains an active data base of document descriptions, as a data structure in the LISP system. The *printout program* draws on that data base in preparing the printout. The system also includes a number of other programs: a *data entry program* that prompts the user for contributions to the data base, a *saving program* that transfers the active data base to a "passive" representation as a text file, and a *re-creation program* that reloads the active data base from one or more such text files. Finally, the data entry program continuously adds the user's input to another text file as a safeguard against the eventuality of system breakdown, and a *recovery* program reloads the text file in that event.

Thus the topmost structure of the system is that there are a number of "data pools", and a number of programs which transfer data between these data pools. (That is of course a third way of program-to-program communication, besides direct calls and data-driven calls). The data flow structure is illustrated in Figure 6.

In the active data base, each document is represented as an atom, with associated properties for author, title, etc. The atom which serves as document i.d. actually looks like B135, i.e., it is essentially a number. Also, each document id has a property which specifies its exact type, which can be either of (presently) six atoms such as BOOK, JOURNAL-ARTICLE, REPORT, etc. Finally, the type name is associated with information as to what properties objects of that type can have: all documents have an AUTHOR and a TITLE property, but only articles in journals and collection volumes have an associated page number.

The major programs in the bibliography system are organized approximately as follows: on the top level there is a loop over a number of document descriptions. In each cycle of the loop, the program determines the type of the document, by prompting the user (in the case of the data entry program), by looking up its TYPE property, or by having it available in the computational context. It then makes a loop over the names of properties that objects of that type can

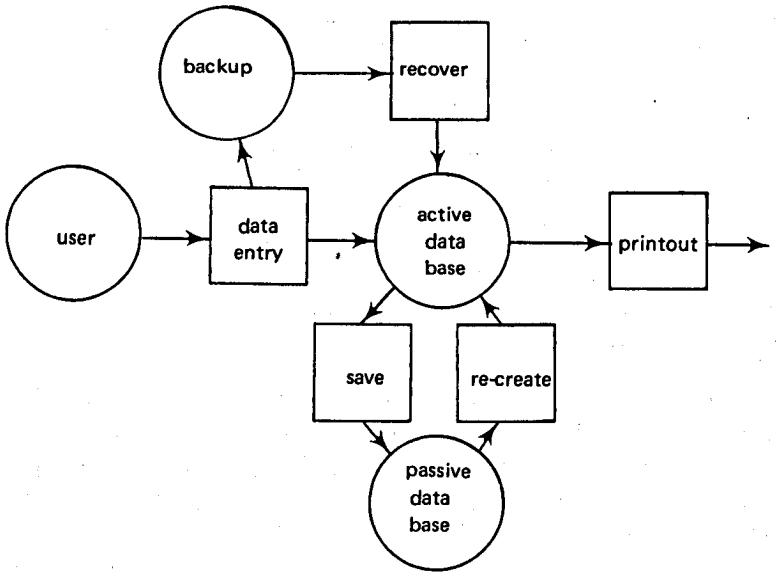


FIG. 6

have, and for each property-name, calls a procedure which is associated with it. Thus a property-name such as AUTHOR is associated with one READFN for prompting the user about that property, one PRINTFN for printing it out, and so on.

I have here idealized the structure for the purpose of simplicity. In actual fact, the "procedure" is often a structure which contains both a procedure name and a number of flags or parameters which have a special significance in the performance of the task. Also, not all programs proceed through the type name to get to the property names: it is sometimes possible to go directly from the document name to the set of properties that it has, make a loop over the existing properties, and call the appropriate procedure for each of them. Such variations to the theme will be further discussed in Part III of this paper, but they indicate that there is a lot of freedom in how data-drivenness is implemented. Data-driven-ness is a style of programming, not one primitive operation (although certain primitive facilities are necessary for doing it). The data-driving or dispatching mechanism is very similar to indirect addressing in machine language. The case construction in higher-level languages serves some of the same purpose, but the dispatching mechanism has the advantage that additional cases can be added dynamically during an interactive session with the programming system, or by another program.

Just as the LISP function *get* can be generalized from the case where a pair of atoms is associated with an expression, to the case where the argument is an arbitrary atom or nested tuple, so also the data-driven procedures can be associated with arbitrary expressions, and not just pairs such as <AUTHOR, READFN>.

The method of data-driven programs is a sound programming practice for a number of reasons. Most of them have been discussed in a previous paper (Sandewall, 1975), but I shall shortly reiterate them here.

Sound naming. Conventional procedures are given names which are "mnemonic" in the sense that when one sees the name, he may get some feeling for what the procedure does. The reverse is however usually not true: if one looks for a procedure that does a certain thing, he is probably not able to guess its exact name. Data-driven procedures are characterized by expressions which are combinations of two or more "names", for example the just discussed <AUTHOR,READFN>. Given some conventions which hold throughout the user's program, such a combination can uniquely specify the purpose of the procedure.

The following example illustrates the point. A few years ago, I wrote a semantic-data-base program, which performs deductive storage and retrieval of typical natural-language expressions (kernel sentences, property assignments, time and space information, etc.) in a data base. The program was data-driven and organized around a number of predicate-calculus relations and functions, about 30 altogether. It used the PCDB structure described above, so for each relation and function, there were a number of procedures for storage of the relationship in the data base, retrieval if the relationship is explicitly stored, retrieval by deduction for open and closed questions, etc. There were altogether about 150 procedures with an almost arbitrary calling structure—one procedure called several others, and there was no visible clustering in the calling structure.

Normally a collection of 150 arbitrary procedures would be fairly difficult to keep track of and update. In this case however, each procedure was characterized by its relation name, its purpose (storage, retrieval, etc.), and in the case of open questions, which argument position(s) were being asked for. As a result it is trivial to find the procedure which performs a given task, and when the program is to be modified there is rarely any question as to where the change is to go, and whether the change may obstruct other parts of the program. And most important, this is achieved without separate documentation of all those procedures—one concise description of the naming conventions is sufficient. Another reason for the clarity of the organization in that program, is that the underlying predicate-calculus representation provided a structure around which the program could be built. One may view it as a relatively problem-oriented description of the task (more problem-oriented than the actual program, that is), and pieces of program were then associated with items in the task description, namely relation and function names.

If the only purpose is to assign structured procedure names, then that may of course also be achieved by "hyphenated" names. Thus the procedure which for a given x determines the y for which SUBPART(x,y) holds, could be called SUBPART_SEARCH_2(x). But it is preferable to let the components of the name be separate entities, so that one can associate information with each constituent (for example with relation names), and also so that driving procedures

can keep one part constant and let the other be variable. This is used in the aforementioned definition of *store* as

store(r,x,y . . .) = get(r,STOREFN)(x,y, . . .)

and of course in many cases in the bibliography system.

Provides an extensible input language. When a program is data-driven using input from the user, the name of each data-driven procedure uses terms which appear in the input language. Thus the agreement between the terminology of the application and the naming in the program, is maximized. This is a desirable practice just for the purpose of manual program maintenance, but it offers the additional possibility of associating both a description of the task environment, and the program for performing a part of that task, with the same atom or "node" in the data base, which makes it easier to check them against each other, or to generate the program from the description. It also makes it maximally easy to extend the input language: new terms are added by adding to the data base a procedure which accounts for that term (of course within the limits and the framework of the executive level of the data-driven program).

Embedded, specialized programming languages. Interpreters, which inspect a program and execute procedures associated with operators in the language, are a special case of dispatching programs. It is common practice in LISP to set up specialized "languages" for specific purposes. For example, the INTERLISP system (Teitelman, 1974) enables the user to choose names for groups of procedures, global variables, and other global data that constitute a module, and which are to be printed out together as a text file. Each "file name" (module name) is then associated with a data structure which specifies how to print the file, for example which procedures shall go on it. In looking at such examples, one finds a spectrum from mere sets of parameters, to expressions in full-fledged programming languages, and the distinction is not particularly interesting. The point is that in order to keep such specialized programming languages small and simple, one wants to be able to call back to the host programming system from them, i.e., to reference procedures or code in the host language from the specialized language. Thus the specialized language can rely on the host system for the assortment of facilities that are always needed, such as file handling and interaction. Embedded languages in this sense seem as the best way to achieve the goal of "extensible programming languages", i.e., to enable each user to tailor the system to fit his needs.

The interpreters that implement such languages must make data-driven procedure calls. A number of programming-language mechanisms which have been proposed in recent years, such as pattern-directed invocation and demons, also rely for their implementation on the method of data-driven programs. Maybe the relative success of such systems is because they made the benefits of data-driven programming available to people who would otherwise not have used it. But it must be better to encourage the user to use the general method, than just provide him with specialized packages for a few operations.

Advising

The following practice is often used by LISP programmers, but may be less obvious to the user of another language. Suppose one wants to organize his program as a set of rules, each of which contains at least a criterium for when it is to be applied, and what one is to do then. Suppose in particular that the criterium for application can be characterized as a piece of data, for example a pattern. One wants to be able to associate several procedures with the same triggering datum, or invocation condition. Therefore, one lets the system create a program skeleton which is the default assignment to each triggering datum, and which may be for example an empty **begin - end** block (or in LISP terms, *progn* form). When the user enters a rule, usually during his interaction with the system, he uses a procedure which inserts the body of the rule in the appropriate place in that structure. In general, the skeleton provides the "glue" or the control structure which keeps the rules together. Thus several rules that trigger from the same datum can be entered at different times, and will be gradually assembled into a procedure.

Another advantage (besides the incrementality) is that the structure of the system as viewed by the user, may be different from the structure as viewed by the executing programming system. Several rules that trigger off the same datum must be kept together during execution, for obvious reasons, but when the user works with his program = set of rules, he may want to group them differently. Advising enables him to do exactly that.

Insertive programming

Both data-driven programming and advising serve the fundamental purpose of modularity: the program is split up into modules which are well named, which can be located in logically appropriate places, and whose interrelationships with their execution environment are well defined and understood. If the modules satisfy these requirements and if they are sufficiently small, such as one or few pages, then one does not have to be much concerned about their insides. Any competent programmer can go into such a small independent program, understand it, and modify it to his needs.

But it may not always be possible to reduce the problem to such small modules while retaining control of the relationships between modules, which raises the problem of program structure within a module, in order to allow larger modules. An obvious candidate is then the *hierarchical* program structure, as argued for example by (Dahl, 1972, p. 176 ff). I have some reservations about organizing an entire program system in a *uniformly* hierarchical fashion (for example, when every level is a sequence of "steps" which are decomposed as the next lower level), but it is clear that a uniform hierarchy in that sense is sometimes a powerful way of organizing a program.

The purpose of this section is to describe an extended program model which I shall call *insertive* programs (as compared to hierarchical programs), and which may be viewed as the generalization of advising to operate in the context of a

hierarchical program. Insertive programs can best be introduced by means of the programming method that goes with them. This is natural since the hierarchical program structure is also associated with programming methods; they can be composed using a top-down method (successive decomposition) or a bottom-up method (successive agglomeration). Wirth (Wirth, 1973, p. 126), says about this. "In practice, the development of a program can never be performed either in a strictly top-down or a pure-bottom-up direction. In general, however, the top-down approach is dominant, when a new algorithm is conceived. . ." But if one studies programs which are developed in this fashion, there appears to be yet another operation, which I shall call *amendment*, where one modifies code *within* one level of the hierarchy.

Consider the following example: we are designing a program that does a certain numerical computation repeatedly for a certain set of values. We have therefore decided to make a loop whose body performs the computation for one value. At the present level of decomposition, we have specified "do the computation" as one operation in the loop, but we have not written out the details of that computation.

We now decide to handle an additional requirement on the program, namely that the sum of the results from all the computations is to be obtained. We therefore decide on a variable to hold the accumulated sum, we declare it at the beginning of the program or block, we initialize it to zero before the loop, update it within the loop, and finally use the sum (for example print it out, or send it to the next computation), after the end of the loop. All of these are of course abstract steps, which may have to be further decomposed. These changes in the program to achieve one single goal, together constitute one amendment.

Other examples of alternating decomposition and amendment (in this sense) are easily found. For example, example 15.2 in Wirth's *Systematic Programming* (p. 133) combines these two operations. The shift from version 3 to version 4 is one example of amendment rather than decomposition.

The purpose of amendments is not to correct errors that have been committed earlier in the design process, but instead to satisfy one additional requirement on the program (as in the summation example), or to improve the efficiency of the program (as in the quoted example by Wirth). Each amendment is conceptually one single thing that one wants to *do*, but it may result in changes in several different places in the program. It is desirable that amendments are done at the "right" time in the decomposition process: if they are attempted too early, it may be impossible to do them, or one is tempted to put the inserts in the wrong places; if they are attempted too late, it may be hard to see where the inserts are best located.

Consider now the obvious environment for this programming process, where the programmer sits at a console and specifies first the top-level structure, and then the successive decompositions and amendments to the programming system. The simplest implementation is to consider this as a case of text editing, and actually perform the substitutions during decomposition, and the inserts

during amendments. In this case the history of the program development is lost. There are however advantages to the alternative scheme where the system retains the development history, i.e., during decomposition it retains both the name of each step in the algorithm, and its decomposition; and when one amendment causes several inserts into the program, it retains the amendment as a separate entity, with pointers between it and the places in the program where the inserts are to go.

In the ideal decomposition case, where several consecutive steps in the algorithm are decomposed independently and in arbitrary order, this scheme allows one to do each decomposition in the textual framework of the surrounding higher-level steps, but without the tedious details of the decompositions of other steps on the same level. Also, this scheme allows printouts or other presentations of the algorithm where different branches in the hierarchy are represented to different depth.

In the simple amendment case (without regard to decomposition) one advantage with this scheme is for modification. If we later want to remove an amendment in the program, all the data that pertain to that amendment are referenced from one single place. Another advantage is for presentation. Suppose a program has been developed in this fashion, and there are a considerable number of inserts into the skeleton. It then becomes possible to just print out the skeleton and one or two sets of inserts, in order to focus one's attention on them.

A further advantage is for defaulting. If we have a "node" or conceptual entity in the system for this summation operation, then that node should clearly contain a reference to the general concept of summation. The general concept may then provide default information about which inserts are necessary, and what their form is likely to be–for example, that the summation variable is usually initialized to zero.

Finally, for cases of combined decomposition and amendment, this scheme allows one to perform the amendment on the appropriate decomposition level of the program, so one does not have to see the lower levels of decomposition that may be irrelevant to the amendment.

The essence of the argument is that it is sometimes useful to consider a program as a combination of a "skeleton" which has been obtained by successive decomposition, and a set of "amendments". Each amendment is thought about by the user as "one thing the program has to do for the user" or "one trick done by the program", but from the point of view of the actual program, an amendment is a set of inserts into the skeleton, each insert being a statement or "step" in the final program. We use the term insertive program for a program with that structure.

Unlike data-driven programming and advising, the proposed method of insertive programming does not seem to be in current use. It is also not trivial to start experimenting with it, since it is hard to administrate an insertive program without the support of a suitable programming system. By consequence, it is also hard to evaluate the method except by asserting its intuitive appeal. But the

method is "in the air" in the sense that some current work on automatic programming (see e.g., Rich and Shrobe, 1975) analyze a user-written program and extract this kind of structure. The proposal here represents a change of approach since I think the skeleton/insert structure can best be obtained as a side-effect of interaction with the user, but the internal representation may be partly similar in both cases.

The idea to "factor out" amendments and represent them outside the main program structure, rather than immerse them in the program, immediately generalizes to a number of other situations. A trivial example is for declarations, which should be associated with a block as an entity, and not be thought of as textually located at the beginning of the block.

Data flow between statements. Still another, and less trivial candidate for factoring-out is for data flow. Consider again an example from the bibliography system. When the data entry program prompts a user for the description of a document, it performs basically the following operations:

- generate an internal name for the document (a "gensym" atom)
- prompt the user for the properties of that document (which involves a loop over its desired properties)
- perform cross-referencing (involving inversion of some of the properties, i.e., the creation of back-pointers from the property to the document name, and also some other construction of new properties)
- save the description of the document on the back-up file

The most natural description of the successive steps have that form, saying "do A, and do B, and do C, . . .". When the steps are implemented in the program, one important thing has to be added, namely provisions for data flow between the steps. In the present program, the atom that is generated in the first step is provided as an argument to the following steps. The second step attaches properties to that atom, and the third and fourth step use those properties.

This data flow can be realized as a program in several different ways. One method is

```
v := generate_name()
prompt(v)
cross_index(v)
save_backup(v)
```

Another method, which assumes that the middle two procedures return their argument as their value, is

```
save_backup(cross_index(prompt(generate_name())))
```

In more complex cases one may want to let one procedure send several values to one or more other procedures, which must be accomplished by packing

several values into a list, or by assigning some or all of them to relatively global variables. Different methods have one thing in common: they clobber up the nice and understandable structure that one had before. Therefore it is here again natural to factor out the data-flow information from the program itself, and specify it in a separate place. At least in routine cases, the programming system should be able to take the responsibility for choosing the appropriate realization of the flow in terms of procedure calls, auxiliary variables, and so on.

The two program fragments above contain only an incomplete description of the real data flow that takes place, since they do not explain that certain properties are assigned (i.e., facts are stored in the data base) in one statement, and accessed in later statements. If the data flow is anyway explicitly represented by statements outside the program, then it would be desirable to describe such data flow via the data base as well (at least) for the purpose of documentation.

Terry Winograd has suggested one more reason for separate specification of data flow (in a private discussion): many common algorithms potentially provide several outputs, but often only some of them are needed. An algorithm to compute the standard deviation may also provide the mean. An iterative computation may return both a result and an estimate of its error. A hashing algorithm may return both the desired result, and an indication that it is time to extend the hash table. The integer "divide" operation returns both a remainder and a quotient. In such cases, one would like to write a general call to the algorithm, and to specify through separate data flow statements which of the outputs are desired, and where they are to be sent.

Direction: conceptual program structure

The original view of a program (for example in machine language) is a coherent document or text whose structure directly reflects the order in which it is executed. Declarations and procedures relax that structure. The methods and ideas described here continue the trend towards the distintegration of that program structure. In data-driven programs, pieces of code are associated with data items that may be part of the description of the data structure (type names and property names in the bibliography system), or of a formalized description of the problem (the PCDB example). In advising, the program is organized as a number of modules which the user may group in any way he pleases. In the proposed insertive programs, major parts of the program body are primarily attached to task-oriented or data-oriented entities.

The logical extension of this trend has been characterized by Charles Rich at M.I.T. as a "raisin-in-the-cake" system. The executable program consists of a number of fragments (the raisins) which are embedded in a structure of declarative or problem-descriptive information (the cake). A part of the system's knowledge is represented through the code in the "raisins", but a considerable part is also represented as the "cake", the threads that lead to a piece of code and cause the system to execute it at appropriate times.

Terry Winograd (Winograd, 1974), in the fifth of his "lectures on artificial intelligence", describes a similar ideal, and uses the term *conceptual program-*

ming. This is a well chosen term: it suggests both that the program is organized around concepts used in a model of the application, and that pieces of coherent code are represented in terms of underlying program concepts (such as the proposed inserts in insertive programming), rather than the conventional surface-structure program. I have therefore used the same term for the title of this paper.

Conceptual programs might not be appropriate for all purposes. If one legislates that important programs must be checked and authorized by a "data ombudsman" or a board of data processing, for example to safeguard the privacy of data, then one certainly wants a closed form of a program which can be inspected, authorized and locked up in a safe for reference purposes. The representation of the program as a document will then be the most appropriate for the foreseeable future. Conceptual programming would probably be used first in experimental-programming and pilot-system situations, where a program is developed in order to improve the understanding of a problem, rather than for actually being used.

In one particular case it is already common practice to use several different "projections" of the same program structure, namely when we use cross-reference list generators. The situation is then that we have a representation of the program (namely the listing) in which some items which we want to see together (namely all occurrences of the same symbol) do not stand close together. The purpose of the cross-reference generator is to transfer the program to another projection, where the logical proximity is physically apparent. The point with conceptual programming is that one should instead attempt to create the underlying conceptual structure, of which both the conventional listing and the cross-reference table are projections (Fig. 7).

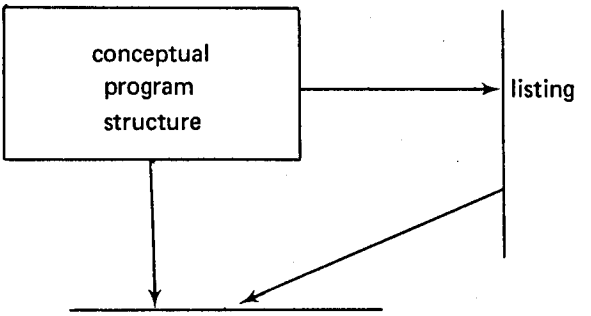


FIG. 7

PART III

A STUDY OF THE BIBLIOGRAPHY SYSTEM

"Conceptual programming" as described in the preceding chapter is an ideal

rather than a method: it represents a plausible direction of research, but not a definite method that can be tried and evaluated. One purpose of the present study was to make the proposal more specific, with regard to the programming methodology and also the supporting programming system.

One way of achieving that purpose might have been to start building a system and see where it would lead, in the tradition of experimental programming. That method was rejected since the resulting system could easily become overloaded with features which were introduced on grounds of generality, aesthetics, or accidence, but which were not really needed. I wanted to know which facilities would really be useful. Two alternative methods were, either to do paper experiments with program development while *pretending* to have a suitable programming system, or to take an actual program, already written by someone else, and try to learn something from it.

Of those two alternatives I preferred the latter, mostly because it offered an opportunity to work with a program of realistic size. A paper experiment could probably not be pushed beyond the level of a toy program of one or a few pages, and I am not convinced that what one can learn from such small programs has an application on programs of more realistic size and complexity. Also, if a paper experiment was to be conducted, it seemed appropriate to first work with an actual program, and then redevelop a program for the same purpose, using the proposed techniques, as the paper experiment.

Dave McDonald's bibliography system, which was described in the last section, was chosen as the object of study. It is highly parametrized and uses data-driven procedures in several ways. and I believe that data-driven-ness is a good first step in the direction of conceptual programming. The present section reports on the observations that were made in the program, and the suggestions that the program implied for how the simple idea of data-driven-ness can be extended, and how a conceptual programming system should be designed and used.

Although the material in this section orginates from the study of one concrete program, it is not intended as an empirical study. Observations made on one single program can of course not really be used as empirical evidence for anything. It is rather a set of suggestions that are claimed to have not merely intuitive appeal, but also a certain concreteness and applicability derived from a contact with a practical program.

Most of the phenomena that will here be described as "observations made in the bibliography program" are fairly standard programming techniques in the LISP-based artificial intelligence community. I believe that roughly the same observations could have been made in another program, or in a program for the same purpose written by another person. However, usually these techniques are implicit: they are used, but never explicitly described or discussed. It was only when I forced myself to study one particular program in depth, that I was able to articulate the methods that are actually being used. When the present paper repeatedly talks about "observations made in the study of the bibliography

program", one should realize that these observations did not involve much surprise.

The order of presentation of the observations and conclusions will be mostly historical, particularly since some observations build on others. As a consequence there is at the time a progression from the fairly obvious to the more significant. Experienced LISP programmers may want to skim lightly over the next few pages and start reading more carefully at the data-structure model (page 248).

The first step was to study the dispatching mechanisms that the bibliography system used, i.e., the chains of references which would lead a calling program through data items to procedures to be executed.

In some cases, the dispatching mechanism was simple and straight-forward, and fit perfectly to the preconceptions presented in Part II. For example, each type name (for types of documents) is associated with a procedure which is executed each time a document description of that type has been entered.

Aggregates

In two interesting cases, the driving data had a non-trivial structure, and seemed to require a *dual description*, namely both as a program in a specialized programming language (in the sense discussed on page 236), and as an aggregated structure that contains a number of data entries. That dual description seemed like it would be a concrete problem for a conceptual programming system.

Let us first describe an example of this, which appears in the printout program, that is the program that produces a nice-looking printout of the properties of a sequence of documents (for use as the "references" section at the end of a paper). The choice of which properties to print out is determined by the type of document (book, article in journal, etc.), and the format is to be variable to allow for different conventions. The bibliography system does that as follows: each set of conventions is assigned a name, which of course is an atom, and which is called a *recipe-name*. The recipe-name has a property under the property-name OUTPUT-RECIPE; whose structure is

```
((pn  params  pnct)
 (pn  params pnct)
 . . .
 (pn  params  pnct)
 ((type  (pn params pnct) (pn params pnct) . . . )
  (type  (pn params pnct)  (pn params pnct) . . . )
 . . . ))
```

Here each *pn* is a property-name for document id's, for example AUTHOR and TITLE; *params* is a sequence of flags which signal specific conventions (for example whether the authors' first names go before or after the last names); *pnct* specifies how to terminate the current property (with a comma, full stop, carriage return, etc.), and *type* is a document sub-type such as BOOK. The printout

program interprets output-recipes as follows: it scans the top-level list, and for each triple of the form <pn params pnct>, it looks up and executes a procedure stored as get(pn, PRINT-UP-FN). That procedure has access to the current document id (so that it can look up its property under the property-name *pn*), and to *params* and *pnct* which specify some details of its operation. When the scan arrives to the sub-lists marked with type names, it chooses the sub-list marked by the type of the present document, and proceeds to scan down its list of triples, but ignores the other branches. Thus the end of the structure serves as an implicit case statement.

There are two ways of thinking about such output recipes, namely as a program or as a location structure (a structure that contains locations that the user can put things in). First, consider the person who has received the instructions for authors in a certain journal, and wants to encode them for the bibliography system in its present operating environment (source code on text files, etc.). That user will want to write up one recipe for that journal, and will be inclined to think of the recipe as a program, which in each step "does" a printout of one item in the document description. For example, he would be inclined to welcome a facility for insertion of arbitrary LISP code in the middle of a recipe which enables him to do "what he wants".

On the other hand, suppose the bibliography system is embedded in a conceptual programming system which "understands" some of its structure, and suppose the user has just told the system that he wants documents to have one more property, namely an NTIS number. The system should then lead the user by the hand and ask him for additional information that the system needs to know, for example how such numbers are to be prompted, and printed out. The system must therefore have a model that enables it to tell what information has to be added where.

It seems to me that such a model could best start with saying that the actual data structure implements a number of mappings, namely

PROPS: type-name → sequence of property-names
TOPRINT: recipe-name * property-name → <params, interpunction>
PRINTFN: property-name → procedure

It should also know how those mappings are implemented: the last one is stored in the trivial way as a property; the first two are stored together in a somewhat non-trivial manner in the output-recipes. But the mappings would be useful as a backbone structure to which one can attach additional information e.g., about how the mappings interrelate, and the first mapping could be shared with other parts of the system, since it is really an ordinary declaration.

This was one of two examples in the bibliography system, and an additional very similar example is provided by the file description mechanism in the INTERLISP system. Let us use the term *aggregate* for a structure which one sometimes wants to treat as a program in a specialized sub-language, and sometimes as a data structure that contains a number of data items which the user

will want to address and modify individually.

The problem that aggregates pose for a conceptual programming system is that both views of the aggregate must be "sugared" or explained to the user, and the system must therefore be able to move freely between the two views. If the user is to think of the aggregate as a program (for example to input an input-recipe), then the syntax and semantics of the program must be explained to him. If he wants to think of it as a composite data structure, then he wants to communicate with the system in terms of underlying structures such as the mappings shown above. If the user is working with one representation and wants to change it, the system should be able to implement that change in the user interface to the other projection, or at least ask the user the right questions about it. This means in particular that the system must know for each projection both how to talk to the user in its terms, and to understand the user when he uses the terms of that projection.

With the data-structure view of aggregates, one would hypothesize a "pure" representation which more directly corresponds to the mappings, and an "aggregation" operation where the composite structure is constructed. At least two reasons for aggregation can be seen in the bibliography program: execution speed (if one knows that the contents of a number of locations will be inspected together, then it makes sense to form a composite structure such as the output-recipes and store them all in there), and presentation for the user (who sometimes may want to see them together, and therefore may be inclined to group them together on the listing of the system).

Another example of aggregation. The bibliography system contains one more example of aggregation, which will be described shortly for the completeness of the record. The data entry program (which prompts the user for the properties of successive documents) assumes that each type name (for sub-types of documents, for example BOOK) is associated with a list of triples of the form

<<prompt-word, explanatory-phrase, property-name>...>

In prompting the user for one document, it first asks about its type and looks up the corresponding list, and then for each triple in the list it prints out the prompt-word, uses the explanatory-phrase if the user requests *help*, stores the input as the property of the document associated with the property-name, and also executes a data entry procedure which may be associated with the property-name. The following abstract underlying structure is natural:

PROPS: type-name → sequence of property-names
TOREAD: property-name → <procedure, prompt-word, explanatory-phrase>

Again the aggregation chosen in the bibliography system is convenient when the user reads and edits a text file of the program, and allows fast execution, but a conceptual programming system should also know the underlying structure.

Dynamic modification of aggregates; self-modifying programs

A second observation was the usefulness of dynamic modification of aggregates. If the aggregate is viewed as a data structure that one makes a loop over, then this is just dynamic looping, where the body of a **for**-loop changes the value of the loop counter. If the aggregate is viewed as a program, it instead amounts to having self-modifying programs. The latter view is interesting considering the present apparently universal belief that self-modifying programs represented a primitive stage in the development of machine language, and that they are antithetical to well structured programs. The observations here provide a counter-example.

Consider the feature whereby the reading program may default some properties. The properties for "reports" (technical reports, internal memoranda) include INSTITUTE (where issued) and LOCATION (the city where the document was published, which usually is the city where the issuing institute is located). Sometimes the system may already know the location of the institute, and it then does not have to prompt the user for it. Several similar defaults are possible. Such defaulting is handled by the bibliography system as follows: each time a document description is entered, a fresh copy of the "prompt-list" aggregate is created. It may contain

$$<\ldots<\text{INSTITUTE}\ldots>,<\text{LOCATION}\ldots>\ldots>$$

A counter steps down that list as successive properties are prompted, and executes the procedures associated with property-names as described above. The procedure associated with the INSTITUTE property tries to default the LOCATION property, and if it is successful, it deletes the triple that would ask for LOCATION, from the current aggregate. (If that is the next triple, it can simply step the counter, otherwise it has to edit the current aggregate). The same method is also used for choosing the right prompt-list for each type: the fresh copy of the aggregate that each document obtains initially starts as

$$<<\text{TYPE}\ldots>,\ldots>$$

and only contains those properties which are common for all types. Thus the first prompt asks for the type of the current document, and the procedure associated with the TYPE prompt adds the specific prompts for the current type, to the current aggregate.

This method is not just a hack! The initial "program" is very readable, since it just says what the system is to ask the user about, and the program modification operations are also very easy to understand, since they can say almost literally, e.g., "remove the question about the LOCATION property". They manage with existing primitives, and do not need to introduce additional symbols or constructs. By comparison, if the same defaulting process is to be performed by a conventional "structured" program, one would have to have a loop over the possible properties, plus a mechanism with boolean flags or other similar devices in order to remember from one cycle to the following which properties are to be

suppressed. In the reverse case where one sometimes wants to *add* prompts chosen from a large number of possible properties, or modify later prompt-words, or the order of the prompts, it is almost impossible to avoid making use of a structure which in some sense is a program, and which is dynamically modified.

Dynamic modification of aggregates adds one more criterium for aggregation: besides improving readability (by clustering the right things together) and improving execution speed, an aggregate of properties which may be used as a self-modifying program, should be designed to contain those things which will change dynamically. For example, if one wants the result of one interaction to be able to modify the prompt-words for subsequent interactions (Hägglund, personal communication, has described a case where that was desirable), then it is necessary to locate promptwords directly in the aggregate, rather than store them as properties of items that appear in the aggregate.

Criteria for the model of the program

Conceptual programming is characterized by the disintegration of the conventional programstructure. The obvious question is then "what structure comes instead". That question was sometimes acute when studying the listings of the bibliography system. When data-driven procedures were involved, one often had to work hard to understand what goes on, and there were also places in the program where a data-driven call would have been natural but had actually been avoided, probably because it would have scattered the information too much (in the listing and/or for the execution of the program). The case construct in the cross-indexing program (described below) is an example.

In the conceptual programming system there should therefore be a structure which keeps track of the miscellaneous data-driven procedures, and is able to explain their location and their purpose to the user. That structure will be a model of the entire system from one particular viewpoint, namely the data-driven calls. There are four major requirements on the model: it should be sufficiently precise that it can be stored in the programming system; sufficiently palatable that the user can read it and enjoy it; sufficient, i.e., contain enough information for its purpose; and minimal, i.e., it should not contain things that it and the user do not need to know. I attempted to extract such a model from the listings of the given bibliography system.

Second-order data-structure models

The first hypothesis was that a second-order data-structure model would be appropriate. The model can best be explained as a generalization of ordinary declarations (in languages which have such). Ordinary declarations for the bibliography systems might say for example "objects of the type JOURNAL shall have the following properties:

property-name	structure of property	purpose
FIRST_YEAR	integer	year of issue of first volume in present series of the journal
SOCIETY	atom	name of the organization which issues the journal
LANGUAGES	list of atoms	official languages of the journal

Such information may be specified formally in a declaration, or informally to the user, and it is sufficiently palatable to be appropriate for the conceptual structure. But if that information is stored in a data base, there must also be specifications about how it is to be stored. What has just been given is then a first-order data-structure model, and one needs a second-order model which says things like: "all objects shall have a *type*. Type names are represented as atoms with the following properties:

property-name	structure of property	purpose
ISA	other type name	type of which this is a specialization
PROPS	list of atoms that are used as property -names	properties that objects of this type may carry

Each object may have properties, with the names specified by the PROPS property of the type of the object. Property-names are represented as atoms with the following properties:

property-name	structure of property	purpose
PROPSTRUC	one of the atoms: INTEGER ATOM LIST-OF-ATOMS	specify the allowable structure of properties with this name

For example, if JOURNAL is a type-name, one may have:

get(JOURNAL,PROPS) = <FIRST-YEAR,SOCIETY,LANGUAGES, . . . >
get(FIRST-YEAR, PROPSTRUC) = INTEGER
get(LANGUAGES, PROPSTRUC) = LIST-OF-ATOMS

The above is only given by way of example, and in actual use one would certainly want to improve on the description. For example "type-name" should itself be made into a type, and so should "structure descriptor" (ATOM, INTEGER, etc.). Also, ISA links must sometimes be handled in a less simple-minded way. The second-order description clearly allows a formal representation. When presented to a human, it tends to be less palatable than the first-order model

because it is more abstract, but that may be compensated with the use of examples.

The conjecture was now that data-driven procedures could be explained to the user by making them part of the data-structure model. The second-order model would then say that there are such things as *types, property-names,* and *structure descriptors*. Among the properties of type names one would find PROPS and as given above, but also a CORELATION-FN property which is a procedure, executed each time a document of that type has been entered. Among the properties of property-names, besides PROPSTRUC above, one may find a READFN property, executed each time a property with that name has been prompted for, a PRINT-UP-FN property used by the printout program, etc. For aggregates, the second-order model would account both for the idealized underlying data structure, and its actual implementation.

Some of the data-driven procedures in the system could actually be accounted for even in a first-order model, namely procedures associated with data items which may appear in input, such as abbreviations. Most of them however required the second-order model.

An immediate observation was that data-driven procedures should not be modeled alone, but together with parametric information which the user could communicate to the system by storing it in the data base, for example prompt words, lists of properties to be written out on files, etc. The distinction between data-driven procedures and parameters is vague anyway, and more important, parameters appeared in the access paths to procedures, for example in the above-mentioned aggregates. The purpose of the model should therefore be characterized as *describing the user-filled locations* in the system, where a *location* is a place which the user can characterize by an atom or combination of atoms, and fill with a parameter or a procedure.

Work on the model was carried fairly far but was not completed. On informal evaluation, the model seemed to satisfy the minimality and the precision requirements fairly well, but it was hard to make it sufficient and palatable. The reason seemed to be that it could not live alone. It is clear that the data-structure model must contain at least informal references to the program structure: in describing the purpose of a data-driven procedure, one would specify which program or procedure it is called by. But it turned out that short informal references were not sufficient; one really had to say fairly much about the program structure expecially in order to explain how to write procedures to put into the data driven-locations.

Program-structure models

The next step was therefore to also develop a program-structure model, which should describe the programs and program parts in the system with respect to their means of communication (data flow, direct calls, and data-driven calls). The top level of the model was a data flow model similar to Figure 6 although somewhat more detailed. Each "program" compartment was then subdivided

into procedures with a procedure-call structure.

Predictably enough, this model required some references to a data-structure model, but it turned out that fairly few references were necessary. When written out as a natural-language text (i.e., candidate program documentation) it specified first the top-level data flow model, then the first-order data-structure model in the major "data pools", and then described each constituent program box. The resulting description turned out to be a good framework for describing the user-filled locations. Essentially each location was described in the context of the program which used it. Only one location was used by more than one program, namely the TYPE property of documents, and it was documented in the introductory data-structure chapter.

Data-structure-based programming

A possible interpretation of these two modelling attempts might be that the program-structure model is in general the "natural" and "primary" one. There is however another possible interpretation, namely that the system *had been written* with a program-structure model literally in the mind of the programmer, and that that is the reason why the formal program-structure model seemed to fit the system better. The discussions with McDonald confirms this: he had thought of the system as a number of programs *with handles on them*, which would enable the skilled user to modify the behaviour of each program, i.e, his model of the system was mostly a program structure.

The data-structure model of the system might then be more appropriate if the system had been designed with a data-structure model in mind. Concretely, one would start out by designing the second-order data-structure model, and the top level of the program-structure model. One would then organize the programs around the data-structure model by working downwards to write data-driven procedures that should be associated with the model, and "upwards" to write the procedures that use the data-structure model and call the data-driven procedures. I shall refer to this method as data-structure-based programming.

Evidence that gives tentative support to data-structure-based programming

There are in fact some designs in the bibliography system that probably would have been done differently and better if that method had been used. On balance, other things might of course have come out worse then, but it is still of some interest to discuss the designs that point in the direction of data-structure-based system development.

Consider the following example. The saving program (for transferring data from the "active" to the "passive" data base) makes a loop over all the properties of each document to be saved, and prints them out. Each property-name, for example AUTHOR, has a dumping procedure associated with it. (These procedures are designed to improve the appearance of the files, in order to facilitate text editing on them). However, since several property-names may use the same procedure, an extra level of indirectness is inserted: the property-name is associated with an atom, which in its turn has a dumping procedure, and several

property-names may share the same such atom.

This common-sense arrangement is easily explained in a program-structure model. But it is interesting to see the names of the intermediate atoms. They are: PNATOM (a term used in the system to characterize an atom in which the distinction between capital and small letters is significant), LIST-OF-PNATOMS, TEXT, and CODE. In other words, one dumping procedure is needed for properties whose structure is to be a list (denoting a set) of atoms, another for properties which are procedures etc. The names of the procedure indicates the structure of the property.

If the second-level data structure model had been developed first, then such structure describing entities would already exist and be ready for attaching procedures to. That design method would then encourage economy of concepts through multiple usage of the same symbol, since the data-structure model is useful in itself (for description and documentation of the system), and also since several different procedures may sometimes be attached to the same symbol.

Another example. The bibliography system offers additional examples that point in the same direction. Here is one: for the purpose of the same saving program, the system maintains a catalogue for each type of document. Since BOOK is a document subtype, there is a global variable BOOKS whose value is a linear list of all document id's of subtype BOOK. That linear list is updated as follows: the data entry program prompts the user for all properties of the current document, and then executes the procedure get(t,CORELATION-FN), where *t* is the type of the document (for example, BOOK). That procedure adds the current document to the appropriate global list. Thus get(BOOK, CORELATION-FN) is

(LAMBDA () (ADDL 'BOOKS REF))

where *ref* is a free variable bound to the current document id, and *addl* does the right thing. Thus again the catalogue names are used only as arbitrary global variables, but they happen to contain information that duplicates the information in the type name. In a data-structure-based system development, one would exploit that correspondence more systematically by making it part of the second-order data-structure model that each subtype of documents shall have a catalogue as its (say) CATALOGUE property. (The property might be the actual list of document id's or the global variable whose value is the list of document id's). Such a decision would make the system more self-documenting. In reading the bibliography program, non-trivial effort was required to find out that there is a correspondence between type names and catalogues, as given by the following table:

type name	catalogue name
BOOK	BOOKS
THESIS	THESISES
REPORT	MEMOS
JOURNAL*	JOURNAL-ARTICLES
COLLECTED	COLLECTED-ARTICLES

*) meaning a paper which was published in a journal.

Finally, with a data-structure-oriented design, one could do away with the CORELATION-FN handle, since its only use in the present system is to increment the catalogue, i.e., to contain the knowledge of the above table.

Dynamic use of locations. The latest mentioned example, about catalogues for document types, also suggests another observation. Our description of "data-driven programming" and the "raisin-in-the-cake" approach in the previous section centered around the *procedure* as the essential component of the system. In forming the data-structure model, it became at once necessary to also include parameters in the model. The latest example indicates that one should also include global variables, or in general, storage locations whose contents are modified during the execution of the program, and which are accessed from such widely different parts of the program that they are considered as global. Therefore, the updated description of data-structure-based system design is that one would start out with a (usually second-order) description of the desired data structure, and when in the course of program development one needs a location to store something in, be it a procedure, a parameter, or global intermediate data, one attempts to choose the name of the location as an entity or a combination of entities that already exist in the data-structure model.

An obvious question is obtained by extrapolation: what about *local* variables or storage locations? There are several obvious reasons for not storing local data in the global data base, and it seems that in most cases one would not gain much with it. If anything, one might let the conceptual system contain definitions of the purpose of some local variables in terms of the global data structure, but other definitions of local variables (in terms of local program structure, data flow, or purposes of code) may often be more useful.

A possible objection to data-structure-based system development is that the suggested ordering is not feasible: one cannot in general define the whole data-structure model first, before one even starts thinking about the program. In our latest example, one might not realize that the catalogue is needed until while one is writing the program. The observation is correct, and the proposal is certainly not that the data-structure model should be designed *and frozen* before the actual programming starts. The proposal is instead that one tentative data-structure model shall be designed at an early stage, and that the program shall be designed "around" it (in the sense described above), but it is evident that if in the course of writing the program one wants to modify the data-structure model, in order to satisfy the needs of the program, he should be able to do so. One obtains the right perspective by thinking of a first-order model as equivalent to the declarations at the beginning of a program in a programming language that uses such, and of a second-order model as a generalization thereof. One tends to write declarations before the body of the program, but one also feels free to modify them when needed.

A third example. The bibliography system did not of course provide any conclusive evidence for or against the utility of data-structure oriented system development. Separate experiments on that issue seem worthwhile, and can be

performed without first developing an advanced conceptual-style programming system. The system did provide one additional example where a data-structure model would have been useful, and that example shall be shortly reviewed here for the completeness of the record, although it does not enable any additional observations. One part of the data entry program has as its major purpose to do "cross-referencing" or property inversion. Thus if the J-NAME property (for journal name, namely the journal in which the paper was published) of the entered document was JACM, then the MEMBERS property of JACM should be augmented with the current document id. That operation is implemented by dispatching on the property name (actually through a construct similar to a case statement, rather than through an actual data-driven procedure call, but that is not essential here), and looks as follows for almost all property-names:

```
(COND ...
     ((EQ TAG 'J-NAME)
      (ADDL 'JOURNALS VALUE)
      (BACKREF))
 ...)
```

or in an Algol-like notation:

```
...  elseif tag = J-NAME
     then begin
          addl(JOURNALS, value);
          backref()
          end ...
```

Here the procedure *backref* performs the property inversion as just described. But the block also serves a second purpose, namely the call to *addl* which adds the current property value (in this case, JACM) to the global value of the variable JOURNALS, if it was not already a member. Thus an atom which appears as a property under the property-name J-NAME is implicitly defined to have the type journal (implying that it can have certain properties), and should be included in its catalogue JOURNALS. Again there is a correspondence table of the form

property-name	catalogue name
J-NAME	JOURNALS
AUTHOR	AUTHORS
UNIVERSITY	UNIVERSITIES
...	...

which is implicitly represented in the code associated with the property-names, as arguments to *addl*. In a data-structure-based design one would be inclined to maintain the catalogues (or catalogue names) as properties of the property-names in the left-hand column. Furthermore, in the actual system the accumu-

lated catalogues are used by a call in the saving program of the form

(FREEZE-DRY '(AUTHORS TITLES JOURNALS . . .))

It would be natural anyway to maintain that argument list in a global location, which then in a data-structure-based design would be just the union of the sets of property-names for all document types. Finally, the procedure *freeze-dry* is actually parametrized with respect to a DUMPPROPS property of catalogues such as JOURNALS; that property is the list of property-names for properties that are to be saved. That item again is a natural part of the second-order data-structure model.

Description of program generation in the program-structure model

Switching back to the program-structure model, it became necessary to describe there not only communication between programs (by data flow and through invocation), but also generation of programs. This was necessary both for the obvious reason that the system assists in the generation of some of its data-driven procedures, but also in order to describe the dynamics of generating and re-loading data files. These points will now be discussed in more detail.

In a conventional data-flow model, one would have two kinds of nodes or boxes, one for "programs" and one for "data". That distinction was not appropriate in the present case, for two reasons. First, there were instances of low-key program generation, where one program would take input from the user and generate appropriate, executable code, which would be called from other parts of the system later during the run. Instead of the program/data dichotomy, it was appropriate to have one single kind of box, but two kinds of arrows that go to a box: an "input" arrow and a "create" arrow. If program box P uses data from box D as input, then there is an arrow from D to P with an "input" attachment to P. If P then generates data in the box D′, there is an arrow from P to D′ with a "create" attachment to D′. In the program-generation case, one box may have both an incoming "create" arrow, incoming "input" arrows, and of course both types of outgoing arrows.

A philosophical problem arises in the representation of procedure calls, for example when generated procedures are called from elsewhere. One would like to consider procedures as special cases of data, and use the same notation both when a program calls a (possibly data-driven) procedure, and when it uses more declarative parametric information provided by the user, particularly since there is a continuum of ways in which the program may "interpret" those parameters. One approach is to define an "inspect" arrow which is used both for procedure calls and for inspection of parameters, and which runs from the caller to the callee, or from a procedure to the parameters it inspects. But then the latter case may be considered as a special case of "input", and could equally well be represented by a data-flow arrow with an "input" attachment–except that the latter was supposed to point in the other direction. One may therefore decide to have just one kind of arrow, for data-flow, and to represent procedure calls as

data flow from the callee to the caller, or one may attempt to distinguish between object-level data flow and inspection of parameters. The former convention is cleaner but takes a while to get used to.

There was also another reason why the program/data dichotomy was inappropriate for the modules in the data-flow model, namely that it was often natural to clump together some kinds of declarative information ("parameters", "data") and some groups of procedures into one entity, and consider the whole module as "parametric". The fact that different parts of the information contained therein related somewhat differently to the LISP interpreter (or compiler) was very irrelevant for the logical clustering.

Files as a special case of programs

The bibliography system also used a less obvious kind of "program generation", namely the generation and subsequent loading of files. Here an explanatory detour is necessary for the reader who is not used to incremental languages. In a conventional language, it is commonplace to have one program which prints out data on a file, for example the current values of some variables, and a corresponding program which reads the file and recreates the same state. The programs must be coordinated: if the printing program produces variable values in a certain order, then the reading program must set them to input data in the same order. In an incremental programming system which performs a read-evaluate-print loop, one usually prefers to let the printing program generate a series of expressions on the file, in such a fashion that the file can be read by the top-level read-evaluate-print loop, and then produces the desired result. Thus if the current value of the variable A is 3, the printing program will print out

```
A := 3
```

or whichever representation of that assignment is used in the programming language. The advantage with that design is that no special reading program is needed, and therefore most of the coordination problem goes away, making it easier to modify the printing program. Also, flexibility can be achieved without loss of transparency. In the conventional print-read pair of programs, one soon gets to the point where the printing program must print out signals or flags which tell the reading program how to handle the subsequent data. Such signals must then be decoded by the reading program, and continued coordination during program changes requires that they must be documented. In the incremental programming system, groups of data are never made to appear "free", but only as arguments to procedures which know what to do with them. If at some time the printing program is extended so that it will print out additional expressions that call a not previously used procedure, then no other code need to be affected.

In a conventional programming environment, with strict distinction between "programs" and "data", an output file should be represented in the program-structure model as a "data" module, which may be the output of one program

and the input of some others. But when the file consists of a sequence of expressions which are to be evaluated, and which contain calls to procedures defined in other program modules then one must consider the file producing program as a program generator, and the program that causes the file to be read, as doing a call to that file-program. The latter observation also gives additional support to the idea of considering any procedure call as a data flow from the callee to the caller.

Summary of program-structure model

The program-structure model of the system was not completely finished, but it proceeded sufficiently far that I was convinced I had seen most of the interesting problems. The design of the model was the obvious one, given what has been said above, and can be summarized as follows: on the top level, there is a data-flow structure between a number of blocks or modules. Each module may contain procedure definitions, data, or both. Arrows represent flow of object-level data, but also generation of and calls to procedures (or groups of procedures). For data-driven procedures, all procedures stored in the same location (for example all procedures stored as getp(p, READFN) for some property-name p) would be in one block, and data-driven calls to them are described like any other calls.

The modules in the top-level model are crudely divided into three groups:

- **program modules**, which could be explained to the user-reader as for example "this is the program that inputs contributions to the data base by prompting the user";
- **parametric modules**, which contain parametric data and data-driven procedures;
- **object-data modules**, which contain actual descriptions of documents in one or another representation (for example property-lists, or as a text file).

As assumed at the outset, the purpose of the program model was to explain not only what the system "does", but also how the user can change the contents of the parametric modules in order to modify the system's behaviour. It therefore referred to a declaration-like description of the object-data modules, and then for each program module described what handles are on it, and how they are used.

In a more exhaustive description of the program, one would also document each procedure in at least the program blocks, specify its purpose, and describe the calling structure within the module. For the limited purpose of documenting the use of parameters, it was found unnecessary in most cases to make such an analysis; in a few cases a limited breakdown of program modules was useful for explaining when the handles would be accessed, and thereby, for explaining how they should be used.

The need for an initialization model

Both the program-structure model and the data-structure model had to be

painstakingly extracted from the listings of the program. The considerable and somewhat unforeseen difficulty of that process raised another question: how do these models relate to the program listing? By what mechanism can one relate items in the description, to items in the listing?

One might have expected this problem to be trivial, on the following grounds: the purpose of the listing is that when it is loaded into the system, it should fill the locations which will be needed when the system is run. Here "location" is used in the sense defined above, i.e., a procedure name, or a property or other place where one item (a parameter, a data-driven procedure) is stored. The listing should therefore consist of a set of expressions, each of which fills one location, and it should be divided into modules that correspond to the modules given in the program-structure model.

Unfortunately things are not that easy. Both in the bibliography system and in many other cases, the programmer organizes his files and the process of assigning locations in a more sophisticated way, for example as follows:

- For some types of parameters and data-driven procedures, one wants to define auxiliary entry procedures, and then use them for input of parameters or procedures. The auxiliary procedures may serve to provide a more compact input (by eliminating repetitions of the same items), and/or to improve the legibility of the text file. When such auxiliary procedures are used, they must of course be input to the system before the expressions that call them.
- Sometimes a group of parameters and/or procedure definitions are separated into their own file, for example in order to variablize over users, or to facilitate restart after certain kinds of crashes, where only some of the locations need to be re-initialized.
- Sometimes locations are not filled when the files are read, but instead the files contain the definition of specific initialization procedures, which are to be called by the user after the files have been loaded. This arrangement is another way to facilitate restart after crashes, but may also be used in order to arrange that all initializations happen in the right order, although they are given in the "wrong" order on the file (in order to make the file easier to work with).
- Sometimes the initialization procedure is defined in one file, and later called from the same or another file, but it may be recalled by the user after a crash.
- Sometimes one location or group of locations may be initialized in several different ways, from several procedures or files.
- Sometimes the main program that the user has to relate to (for example the prompting program) performs its own initializations when it is first called, before it goes into the interactive loop. Some-

> times it behaves differently the second time it is called, so the first time it must make a note that it has been called, to be seen the second time; but then if a more thorough restart has been done inbetween (for example by reloading certain files) then that note must be overwritten. And so forth.

In summary, the listings of the system cannot be viewed simply as a set of assignments of contents to locations. A better approximation is to say that the files are together a program whose purpose it is to fill those locations. But even that description is insufficient because of the initializations that take place after the files have been loaded.

Proposal for an initialization model. A sufficient model must start out by separating two questions: first, how is the system loaded, meaning how are locations initialized with their contents, and second, how is the system run, using the contents of the locations. (The distinction may not be aboslute but I believe it is sufficiently clear to be useful). Both questions assume of course a model of what locations there are, i.e., the second-order data-structure model that was proposed earlier. For describing how the system is loaded, one must start with a *description of an interactive session*, for example as

a. load files containing procedures and parameters
b. call initialization procedures
c. call the top level procedure of the system

where the names of the files and procedures must be explicitly stated for each system. Either or both of steps b and c may sometimes be omitted, and additional steps may sometimes be necessary to complete the picture, for example, the execution of the user's entry file which is done when the LISP system is entered.

The description of the interactive session is in fact a program, although it may not be fully specific about the relative order or number of repetitions of certain steps. It may even be data-driven, for example if an initialization procedure wants to load a file, but asks the user for the name of the file, or asks the operating system for the current user id in order to load the correct file. But more important, the session description is similar to declarations in the sense that it is assumed to be relatively constant while the program is debugged and extended. The contents of a file may change, but the name of the file as found in the session description does not need to change except very occasionally.

The purpose of such a session model or loading model wold be to describe how locations are loaded with contents. In order to do that, the loading model must be related to the data-structure model and the program-structure model. The latter relationship is easily done just by extension downwards: the session model contains statements to load a file or call a procedure, and each file and each procedure can again load a file or call a procedure. Thus an ordinary calling-structure diagram is appropriate. Notice however that some parts of the

model may be "less variable" than others. For example, assume in some system that the file INIT is to be loaded first, and that INIT shall contain commands to load certain files, so that the system can be extended by adding more file calls to INIT without telling the user. Then the statement that INIT shall be loaded during the interactive session is "less variable" than the loading commands contained in one generation of INIT. The combination of the initialization model and the program-structure model must account for such distinctions.

Relationships between the initialization model and the data-structure model. At a certain level in the program-structure model for initialization, one will find procedure calls or other similar operations which serve to initialize one or a few locations, and where the arguments in the procedure calls are often constants (or "quoted", to use LISP jargon), namely they are the intended contents of the location, or data which are to be transformed in order to generate those contents. (Complications are of course possible, for example when data are first stored in one location and later retrieved, rearranged, and stored in another location.) Let us call these the elementary initialization operations. Above them in the calling structure, one finds procedure calls which mostly serve the purpose of grouping the elementary initialization operations into natural groups; below that level one finds miscellaneous auxiliary or system-level procedures.

Each elementary initialization operation is then characterized in three ways: in which group (procedure or file) does it occur; which procedure is called there; and which location(s) are filled by this operation.

When describing a system, it seems natural at first to maintain the references in that direction, and in particular, to specify for each elementary operation which locations it fills. But it may be advantageous to maintain the system in the opposite way, i.e., to restrict the initialization model to a "skeleton" (in the sense of the term used in the section on advising, page 237) which extends down to but not including the level of elementary initialization operations, and then for each location maintain a reference to the point in the initialization skeleton where an elementary initialization operation for it is to be located. This would be a special case of advising: if the user decides to establish another location in the system, he specifies not only the ordinary declarative information, but also a pointer to the place in the skeleton where the location is to be initialized. This information is retained and may be used both for specific questions ("This location does not seem to have been initialized. Where was that supposed to have been done?") and general questions ("Has initialization of all locations been accounted for?").

A possible objection is that the same purpose may be achieved with simpler means, for example a conventional cross-index of the text file of the system. But maintaining the pointer from the location name to the initializing operation in the system has the advantage that it enables more informative answers. If the user wonders why a certain location has not been initialized, then the listing of the system plus the cross-reference list will at best provide the information whether the listing contains a statement that is intended to do the initialization. A

programming system that has access to the reference may instead potentially give the answer "that location is supposed to be initialized by file FIE, which is loaded by file FOO, but this time FIE has not been loaded because FOO bombed out before it got that far". For another example, suppose the procedure *storesym* has been defined as

```
storesym(a,r,b) = begin
                    addprop(a,r,b);
                    addprop(b, rev(r), a)
                  end
```

and suppose the user has by oversight arranged his files so that storesym(BOY, SUBSET-OF,MALE) is executed before put(SUBSET-OF,REV,SUPERSET-OF) has been performed. When the user later wonders why get(MALE,SUPERSET-OF) is not defined, the system should be able to explain that to him, if it maintains an initialization model and the knowledge where in the model each location is supposed to be initialized. Clearly these are pointers from the second-order data-structure model, to the session description and its expansion into an initialization model.

It may also be remarked that cross-reference listings are not so appropriate when locations are named by combinations of two or more items, which we have argued is a desirable practice. If one has achieved an economy of names, and if the combination of names is significant, then the cross-reference will tend to generate a bigger "fanout" than if single names had been used, and therefore it will assign an immediate relationship to items which may not be at all related.

We assumed above that a distinction could be made between the process whereby the system is loaded and locations filled, and the process wherein the system is run and the contents of the locations are used. One can of course not expect these two aspects of the program to be physically separated in the code: some initialization may be performed when the execution has already progressed far, for example by default the first time the information is needed. The assumption is merely that at a certain level in the program-structure model, one can make a reasonably clear distinction between initialization operations and execution operations.

In summary, we have now arrived at three interlocked projections or partial models that seem to be useful to have, namely (1) a second-order data-structure model, which describes the system as a location structure; (2) a program-structure model which describes the system as data-flow between top-level modules, and a procedure-call structure that extends downward from there; and finally (3) an initialization model that describes how locations are filled with their contents. Additional projections or models have been proposed (for example the idea of a "purpose flow", due to Rich and Shrobe) but would extend the discussion too far for the present report.

Data base management

Before the paper ends, we should add two short remarks. One concerns the

management of the integrated program/data structure. The proposals in the present section imply that the programming system should contain not only programs and data, but also descriptions of data, descriptions of programs, descriptions of their relationships, and so on. In order to use these proposals in practice, one must generalize conventional program management systems into data base management systems which treat procedures in the data base as a special case. (Sandewall, 1975) describes a first shot at such a data base management system.

The structure of the programming system

A LISP programming system itself takes some steps in the direction of conceptual programming. First, like any other interpreter it is data-driven with respect to names of elementary functions or procedures. But a modern LISP system also contains a large number of "handles" where the user can attach his own code or parameters to modify the system. It would be interesting to attempt to design a programming system using the methods proposed in this paper. Hopefully one would obtain a more consistently organized and more self-documenting system than is usually the case. Moreover, the models of the programming system might be useful for example when the user has to describe an initialization process where certain handles on the system are set in order to prepare ground for the continued initialization, for example when the status of certain characters (such as break characters) is changed.

PART IV

SUMMARY AND CONCLUSIONS

The present work has a strongly inductive character. Given a set of ideas about how one ought to think about a program and about programming, here characterized as "conceptual programming", I have studied one particular program in depth, partly in order to obtain suggestions for how the proposed "conceptual programming" method is to be realized. These are suggestions rather than empirically proven facts, since the study of one particular program would not provide sufficient evidence for the latter. They are also highly interdependent but I shall still venture to make a list of the conclusions.

Conceptual programming is an approach which contains two programming methods which are commonplace among users of LISP and similar languages already today, namely

- data-driven procedures (= dispatching), where procedures are called indirectly via data items;
- automatic advising, where the programmer first writes or generates a "skeleton" code, and then lets other procedures insert pieces of code into the skeleton.

It also contains a style of programming which is not in current use today, since it requires its own kind of programming system, but which seems sufficiently concrete that one could make some experiments with using it, namely

- insertive programming, where the program is represented by a hierarchical structure, plus a number of inserts of various kinds (additional pieces of code, information about data flow, etc.) which are located elsewhere but point into the structure to indicate where they belong at execution time.

In the very long range, conceptual programming is the ideal situation where the program is truly represented as a belief structure in the computer, and the programmer or tutor normally does not work with exhaustive listings of this structure, but only with specific extracts or projections which have been obtained for specific purposes.

The major conclusions which were obtained from the study of the simple program were:

- the location concept, where a location is a globally defined position which is described by a combination of identifiers that are understood by the user, and which may contain parameters, or a procedure, or certain dynamically changed data (for example catalogues).
- the importance of aggregates, i.e., structures which allow a dual description either as programs in very specialized languages, or as composite data structures that contain several locations.
- the usefulness of dynamic modification of aggregates, which may be thought of as self-modification in a program.
- the usefulness of second-order data-structure descriptions, where the information contained in conventional declarations, as well as other information about the same entities, is stored in the data base itself.
- a programming style, here called data-structure-based programming, where one specifies a tentative second-order data-structure model early in the design process, and then attempts to use combinations of entities in that model as names for locations as the program is developed. The purpose of this process is to obtain a system with a clear and self-documenting structure for both data and programs, and also to achieve economy of concepts and minimize the proliferation of "mnemonic" names. The support for this suggestion, besides its possible intuitive attraction, consists of a number of constructs in the sample program which would very likely have been designed in a more systematic way if the proposed programming style had been used.
- the second-order data-structure model is palatable only if it co-exists with a program-structure model, which on the top level (at least for

the present example) describes the data flow between a number of "data pools", and below that level, a calling-structure model.

- the data-flow model at the top of the program-structure model should not distinguish between "programs" and "data", since low-level program generation is common; since the distinction is irrelevant for the natural grouping of procedures and parameters into aggregates and into modules; and since files in this kind of programming system are most appropriately viewed as a kind of program.
- instead of the program/data distinction, there is a signficant distinction between "input" and "create" arrows. A procedure call may be considered as data input from the callee to the caller.
- besides the data-structure and the program-structure models, one also needs an initialization model. This model consists of two parts. The first part is a description of an interactive session, which is a kind of program which may call procedures and/or files (those two are entirely equivalent for this purpose). The session model is a skeleton in the sense of automatic advising. The second part of the initialization model is a set of inserts which for each location in the data-structure model specifies where in the initialization model there should be an operation that initializes this location.

ACKNOWLEDGEMENTS

It should be clear from the foregoing that many thanks are due to Dave McDonald, who let his program be subjected to this exercise, and also took a lot of time to discuss the program as well as the issues addressed in this paper. Several other researchers contributed with comments and ideas, particularly Anders Beckman (Uppsala), Carl Hewitt (MIT), Mats Nordström (Uppsala), Charles Rich, Howard Shrobe and Jerry Sussman (all MIT), and Terry Winograd (Stanford). The work reported here was mainly conducted at the MIT Artificial Intelligence Laboratory and the Stanford Artificial Intelligence Laboratory, both supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense.

REFERENCES

Beckman, L., A. Haraldson, Ö. Oskarsson, and E. Sandewall (1975) A partial evaluator, and its uses as a programming tool. *Internal report DLU 74/34*, Department of Computer Science, Uppsala University, Sweden.

Dahl, O.-J. (1972) Hierarchical program structures, in O.-J. Dahl *et al.*, *Structured Programming*, Academic Press.

Rich, Ch. and H. Shrobe (1974) Understanding LISP programs: Towards a programming apprentice. *Working paper 82*, MIT Artificial Intelligence Lab.

Sandewall, E. (1971) PCDB, a programming tool for management of a predicate-calculus-oriented data base. *Proc IJCAI2*, Computer Society, London.

Sandewall, E. (1973) Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. *Proc IJCAI3*, Stanford Research Institute.

Sandewall, E. (1975) Ideas about management of LISP data bases. *Proc IJCAI4*, MIT.

Teitelman, W. (1974) *INTERLISP Reference Manual*. Xerox Palo Alto Research Center,

Palo Alto, California.
Wegbreit, B. (1974) The treatment of data types in EL1. *CACM* 17.
Winograd, T. (1974) Five lectures on Artificial Intelligence. *Memo No. 246*, Stanford A.I. Laboratory.
Winograd, T. (1974) Breaking the complexity barrier, Stanford A.I. Laboratory.
Wirth, N. (1973) *Systematic Programming, an Introduction*. Prentice-Hall.

15

Programming with Resolution Logic

M. H. van Emden

Department of Computer Science
University of Waterloo, Canada

INTRODUCTION

In the late sixties the newly discovered resolution principle for first-order predicate logic aroused great expectations among workers in Artificial Intelligence. As it turned out later, research in resolution theorem-proving did not at that time have a broad enough scope to support the requirements of constructing 'intelligent' and 'purposeful' computer programs. By about 1970 resolution logic had fallen into disrepute and the emphasis in AI research shifted towards the development of new program languages of very high level intended to be suitable for programming more directly than hitherto possible the 'knowledge', reasoning power, and goal-directed behaviour required by an 'intelligent' computer program.

This development has taken an interesting turn with the more recent discovery of a program language within resolution logic. The new program languages for AI research have many interesting features, but they also give the impression of being rather experimental, *ad hoc* precursors to a single coherent language which will supersede all of them. One is tempted to speculate whether resolution logic might be the basis of such a language.

Whether or not this will be the case, the discovery of resolution logic as a program language is an important development in several respects:

a) resolution logic shares with some of the new program languages for AI research the property of being more descriptive, or of higher level, than conventional procedure-oriented program languages,
b) by specifying algorithms in resolution logic one can use to advantage a separation of an algorithm specification into two distinct components: the 'logic' component and the 'control' component,
c) resolution logic as a program language has a precisely defined and simple semantics,
d) in theoretical studies of computation, resolution logic as a program

language is an attractive alternative to the usual formalisms such as Kleene's recursion equations, Turing machines, the formal grammars of the Chomsky hierarchy, and so on.

In this paper I am mainly concerned with a) and b). As for c), I will just state a syntax and a semantics, because this does not take much space anyway. The tempting possibilities in d) have to be left out altogether.

As for a), the first question to be addressed is whether it is at all possible to use resolution logic as a program language. It seemed to me best to illustrate the affirmative answer with an example of a problem typical of conventional programming: sorting a sequence. The most important thing is to show that a resolution theorem-prover can solve straightforward problems in a straightforward way. The choice of the example should not suggest that I consider sorting a proper problem-domain for program languages of very high level.

The second question to be addressed is in what sense resolution logic is an advance with respect to conventional procedure-oriented program languages. I argue that all advances in programming, the conventional languages included, can be regarded as steps toward automatic programming. This takes me into the slippery questions of why one program language is of 'higher level' than another, or more 'descriptive', or less 'imperative'. I shall argue that this distinction can be made more precise by analyzing specifications of algorithms into a logic component and a control component and that it is precisely the possibility of doing this when programming with resolution logic that contributes to its being an advance in the development of program languages.

THE ROLE OF SPECIFICATION LANGUAGE IN AUTOMATIC PROGRAMMING

Automatic programming means the automation of program writing. To take a specific example, think of writing PL/1 programs, possibly a profitable target of automation in programming. The result of successful automation would be a machine (a "PL/1 machine") writing PL/1 (the *target language*) programs. Such a machine would still have to be told, no matter how automatic otherwise, what the product of its activity is expected to do, for instance in the form of a specification of input-output behaviour. Machines being what they are (for the time being), such a specification would have to be written in a formal language (*specification language*).

This is reminiscent of the existing situation where the writing of machine-code programs has been automated. There exist machines which accept a "specification" in a formal language (for instance PL/1) and produce machine-code programs that are expected to comply with these specifications. This shows that, if understood in a certain way, automatic programming has been going on for a long time already. Its purpose is to produce with less effort better programs. Would a PL/1 machine achieve any progress towards this goal?

The PL/1 machine would operate in an environment (schematically shown in

Figure 1), which would only make sense if specifications can be better written in specification language than in PL/1: the PL/1 machine would act as an interface between specification language and machine code. However, the PL/1 language was intended (insofar as it developed purposefully at all) as an interface between a *human* programmer and machine code; why should it be adopted for the other purpose?

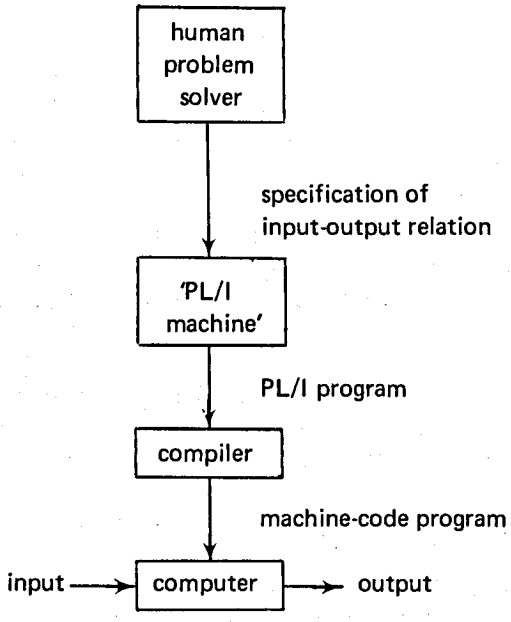


FIG. 1. Environment of the 'PL/I machine'

Either one needs a language intermediate between specification language and machine code and then it seems better not to adopt PL/1, but to start with a clean slate. Or one does not need any intermediate language and PL/1 is itself a candidate specification language. Again, it seems better to start from scratch and to look for a language especially suited for this purpose. Of course, these considerations apply not only to PL/1 but also to other conventional program languages. In either case automatic programming will not turn out to be an unprecedented innovation but a further step towards the use of more powerful programming tools as assemblers, interpreters, and compilers have been in the past.

TWO ASPECTS OF ALGORITHM SPECIFICATION

The preceding observations suggest that there may not exist a clear-cut distinction between a specification language for automatic programming and a higher-level program language. They also suggest that any step towards auto-

matic programming will be one in an ongoing evolution towards more powerful tools for computer-aided problem-solving. Indeed, the pioneers in compiler design already flew the banner of 'Automatic Programming' (Annual Reviews in Automatic Programming, 1960-1970).

Although no clear-cut distinctions will emerge here, it is useful to compare two aspects of algorithm specification: the *imperative* aspect is typical for the lower level of programming as is the *descriptive* aspect for the higher level. In a machine-code program it is spelled out *how* things are done, but it is always very hard to see without additional explanations *what* is being done. This is an extreme case of an imperative specification of an algorithm. At the other extreme, in a specification it is only explained what is to be done and it is the problem of automatic programming to convert this into commands saying how.

Strictly speaking, a language like PL/1 or Algol 60 is completely imperative: every statement corresponds to commands to be executed. However, the value of such a language lies in the fact that in a well-written program it is possible to see without additional explanations what is being done: such a program has descriptive value as well as an imperative effect. Some of the imperative aspects have disappeared from the program, like the details of storage allocation and the commands involved in procedure invocation. The ABSYS language (Elcock, 1968; Foster, 1968; Foster and Elcock, 1969) is an interesting experiment that allows algorithms to be specified in a more descriptive manner. In this respect it was a forerunner of some of the new program languages for AI research (Bobrow and Raphael, 1974).

Predicate logic is usually regarded as a purely descriptive language: at most able to express what is to be done by a program and not how to do it. Yet, with respect to a given proof procedure, a specification in logic has implications for the imperative aspect, as will become clear by comparing with each other the two versions of the sorting example below.

Although the descriptive and imperative aspects of algorithm specification may be hard to disentangle, I think the distinction is useful for characterizing what constitutes a higher level program language: one that has less commitment to the imperative aspects of the algorithms to be specified and, by being more descriptive, is easier to write in and to understand for the human problem-solver.

THE CONTRIBUTIONS OF GREEN AND OF KOWALSKI

Green (Green, 1969) has given a very useful definition of four different tasks in automatic programming by representing them as problems in automatic deduction. He specified the input-output behaviour of the required program as a set A of axioms in first-order predicate logic containing a predicate symbol R such that $A \Rightarrow R(s,t)$ (A logically implies R(s,t)) if and only if the program is to give output t for input s. Thus, A defines (with respect to the predicate symbol R) a relation in the mathematical sense between inputs and outputs.

The generality of relations (as compared to functions as usually studied in mathematics) is suitable here: the required program, as a map from inputs to

outputs, need not be total (an output may not exist for some inputs) and it need not be determinate (an input may be followed by any of more than one possible output). Even if the program computes a total function, there is no disadvantage in specifying it as a relation.

Green distinguishes *checking, simulation, verification,* and *synthesis* as tasks in automatic programming. He shows that an automatic theorem-prover can in principle accomplish these (given a suitable set A of axioms, not necessarily the same for each task) in the process of proving a theorem of a particular form. Figure 3.1 shows how this form determines which of the four tasks is to be carried out.

FORM OF THEOREM TO BE PROVED	POSSIBLE ANSWERS	TASK
R(a,b)	yes no	checking
$\exists x.R(a,x)$	yes, x=b no	simulation
$\forall x.R(x,g(x))$	yes no, x=c	verification (of program g)
$\forall x.\exists y.R(x,y)$	yes, y=f(x) no, x=c	synthesis (of program f)

FIG. 3.1. Green's tasks in automatic programming

Note in this figure that only 'synthesis' corresponds to automatic programming as described above. The specification language is predicate logic and f is the synthesised program in a target language embodied in the function symbols of the specifying axioms. Synthesis appears to be a difficult problem. Before attacking it, let us pause and consider whether there is not a way around.

To find such a way, we should ask: what is the purpose of a program, and can it not be achieved in another way? The answer is, a program is to cause computations to be done automatically on a computer and, yes, it can be done in another way: by simulation. As we see in Figure 3.1, for given input *a*, the automatic theorem-prover will produce the output *b* that would have been generated by the program synthesised from the axioms A. But then, why would we need the synthesised program if, for any input, we can get by simulation the required output without the program?

The possibility is at least worth investigating, although at the time of Green's work it did not seem to be the most promising approach. In order to make simulation a practically interesting possibility, both development of theorem-proving technique and an increased understanding of the pragmatics of predicate logic were needed.

These requirements have been met in the meantime. The SL-resolution proof procedure of Kowalski and Kuehner (Kowalski and Kuehner, 1971), or each of several related proof procedures (Reiter, 1968; Loveland, 1969), can be adapted

to act like a program-language interpreter for a suitably constructed sentence. The 'procedural interpretation' for resolution logic is Kowalski's (Kowalski, 1974a) great contribution to the pragmatics of first-order predicate logic.

Since several years these ideas have been realized in the PROLOG system developed by Colmerauer and his colleagues in the University of Marseilles (Colmerauer, *et al.*, 1972). The action of the PROLOG system is in principle that of the simulation variety of automatic programming according to Green (see Figure 3.1). As will be explained later, the system can also be regarded as an interpreter for a generalized Algol with backtracking.

The use of predicate logic discussed in this paper has some features in common with that of Hayes (Hayes, 1973), who arrives at a program language by adding 'control information' to axioms of logic in order to obtain computationally favourable behaviour from a resolution proof procedure. The work of Hayes suggests a fruitful approach to the study of algorithms: to decompose the specification A of an algorithm into a 'logic' component L (to be specified by a sentence in resolution logic) and a 'control' component C, specified in some other way. Kowalski (Kowalski, 1975) has formulated the decomposition as

$$A = L + C$$

and showed an example of two algorithms whose specifications A_1 and A_2 are related as follows:

$$A_1 = L + C_1 \qquad \text{and} \qquad A_2 = L + C_2$$

That is, the algorithms specified by A_1 and A_2 are different, but only in the control component. In this paper we shall see that A_1 can be a specification of a sorting algorithm according to the 'quicksort' principle and A_2 can be a specification of a permutation generator. Because the control components C_1 and C_2 are comparatively inconspicuous we shall see the interesting phenomenon of a 'quicksort' and a permutation generator with almost the same specification.

The example on sorting has been included to draw attention to the fact that an autonomous resolution proof procedure is capable of computationally acceptable behaviour. This is incompatible with widely-held opinions on this point. To give an example of such an opinion, I quote Hayes (Hayes, 1973):

> "However, there is every evidence, both practical and theoretical, that an autonomous resolution theorem-prover will never be sufficiently powerful to cope with complex problems. The practical evidence is abundant in the literature on computational logic."

I cannot make less abundant the practical evidence here alluded to. What I can do is to add some evidence for the contrary opinion that autonomous resolution proof procedures can be computationally useful. Of course, the examples in this paper do not solve complex problems. They suggest, however, that predicate logic can be used as a high-level program language. Hence an autonomous resolution proof procedure (for instance, Kowalski's LUSH system to be described in

this paper), acting on a sentence in logic which is pragmatically sound according to the procedural interpretation, can cope with problems at least as complex as can be programmed in a high-level program language.

But, as a program language, resolution logic is of a higher level, more descriptive, less imperative, than languages like PL/1 or Algol 60. In fact, the reader may recognize several of the features of the new program languages for AI research (Bobrow and Raphael, 1974). Resolution logic remains a natural language for stating facts and making inferences, in addition to its newly discovered use in effectively simulating the execution of a program. All these considerations suggest that in resolution logic as powerful problem-solvers will be programmed as in any other program language for AI research.

It remains, of course, a requirement that specifications in predicate logic be pragmatically sound according to the procedural interpretation. That is part of the art of programming in logic. It need not be a superhuman achievement to express a complex problem in pragmatically sound axioms. In fact, this has been done in PROLOG with remarkably little fuss for several ambitious programming tasks. These include natural-language understanding systems (Colmerauer, *et al.*, 1972; Paséro, 1973), formula-manipulation and symbolic-integration systems (Bergman and Kanoui, 1973; Kanoui, 1973), and a STRIPS-style problem-solver (Warren, 1974). A comparison has been published (Warren, 1974) between the last-mentioned program and the original STRIPS system. In the examples tried, the PROLOG problem-solver was considerably faster. A more important advantage of PROLOG is suggested by the fact that writing and testing the program required about one man-week.

RESOLUTION LOGIC AS A LANGUAGE FOR STATING PROBLEMS

In general, a syntax for first-order predicate logic comprises a language expressing sentences, an inference system consisting of axioms and rules of inference, and a proof procedure relating the use of the inference system to the sentence to be proved. The usual language and inference system ('classical first-order predicate logic') are found, for instance, in (Kleene, 1967).

For the purpose of automatic deduction a different syntax has evolved. Its language is the 'clausal form' of first-order predicate logic. Its inference system is the one invented by J.A. Robinson (Robinson, 1967), using his 'resolution principle' and his 'unification algorithm'. I will refer to the language of clausal form together with the resolution rule inference as 'resolution logic'.

Resolution logic has been criticized as being unsuitable for problem-solving because of an allegedly inherent lack of goal-directedness in its deductions and for the alleged difficulty, or impossibility, of taking domain-specific knowledge into account for the control of the course of a deduction. Now the control is in the domain of the proof procedure. No particular proof procedure is by necessity associated with resolution logic. The criticisms mentioned above are possibly based on a mistaken belief that proof procedures associated with resolution logic must be of the 'uniform' or 'saturation' variety.

Several researchers (Bledsoe, 1971; Bledsoe, *et al.*, 1972; Kowalski, 1974a; Michie, *et al.*, 1972; Michie and Sibert, 1974; Robinson, 1967) have realized the rich variety of possible proof procedures which can be used in resolution logic. They developed procedures guided by heuristic principles or by the desirability of a procedure being guided by domain-specific knowledge. In this paper I will be concerned with one of the approaches mentioned, namely Kowalski's.

Resolution logic can be explained in terms of classical logic, as is done in (Nilsson, 1971). I believe that resolution logic has advantages other than the one of being suitable for use on a computer: for example its simplicity, which makes it also suitable for a self-contained definition of syntax and semantics. A pragmatics will be given in terms of classical logic.

A language for resolution logic (see Kowalski, 1974a)

A *sentence* is a set of clauses. A *clause* is a set of literals L_i written as

$$L_1 \ldots L_n$$

except when the set is empty: it is then called the *null clause* and is written as

$$\square$$

A *literal* is either a *positive* literal +A or a *negative* literal -A, where A is an atomic formula. An *atomic formula* is written as

$$P(t_1, \ldots, t_m), \quad m = 0,1, \ldots,$$

where P is a predicate symbol and $t_1, \ldots, t_m$ are terms. A *term* is either a variable or an expression $f(t_1, \ldots, t_k)$, k=0,1,. . ., where f is a k-place function symbol and the t_i are terms. A *constant* is a 0-place function symbol. For the sets of *predicate symbols, function symbols,* and *variables* one is free to choose any three mutually disjoint sets of symbols.

In the examples of this paper, a predicate symbol is a word starting with an upper-case letter; a variable is a word starting with a lower-case letter; a constant symbol is a word in italics.

A semantics for resolution logic (see van Emden and Kowalski, 1974)

The semantics of a language determines the meaning of a sentence of the language: it deals with the relationship between a sentence of the language and a universe: a set of objects endowed with a structure which is partially determined by the structure of the language. An interpretation assigns a meaning to the variable-free terms constructible from the function symbols in the sentence and to the predicate symbols in the sentence. The remaining part of semantics determines whether the sentence is true in a given interpretation.

As a structured set of objects to serve as universe I consider here the so-called *Herbrand universe*: the set of all variable-free terms that can be constructed from the constants and other function symbols of the sentence.

The set of all atomic formulas $P(t_1, \ldots, t_k)$ such that the predicate symbol P

occurs in the sentence and such that $t_1,\ldots,t_k$ are in its Herbrand universe, is called the *Herbrand base* $\hat{H}$ of the sentence.

Any subset of I of $\hat{H}$ determines a *Herbrand interpretation* in the following way. A variable-free term denotes itself (note that the term occurs itself in the Herbrand universe). The meaning of an n-place predicate symbol P is the following n-ary relation over the Herbrand universe

$$\{(t_1,\ldots,t_n) : P(t_1,\ldots,t_n) \in I\}$$

The following rules determine whether the sentence is true in the interpretation determined by I. If this is the case, then I is said to be a *model* of the sentence. A sentence is said to be *unsatisfiable* if it has no model.

A variable-free literal +A is true iff $A \in I$.
A variable-free literal -A is true iff $A \notin I$.
A variable-free clause is true iff at least one of its literals is true.
A clause is true iff every one of its variable-free instances is true.
A sentence is true iff each of its clauses is true.

A pragmatics for resolution logic

Semantics determines meaning in a strict sense only involving a relationship between the language and a universe. The sense of meaning that also involves the user of the language, for instance in the form of explanations helping her to understand a sentence, belong to the *pragmatics* of the language.

The concepts of classical predicate logic are intuitively understood by many people; it is therefore useful to express the meaning of sentences of resolution logic in terms of classical logic, as has been done by Kowalski (Kowalski, 1974a) in the following way.

The meaning of a sentence $\{C_1,\ldots,C_n\}$ is the conjunction

C_1 and ... and C_n.

Suppose a clause contains variables $x_1,\ldots,x_k$ and suppose it is written such that no negative literal precedes a positive one. The meaning of the clause

$$+B_1 \ldots +B_m\text{-}A_1 \ldots \text{-}A_n$$

is a universally quantified implication:

for all $x_1,\ldots,x_K$
B_1 or ... or B_m is implied by A_1 and ... and A_n

It may be helpful to have a special reading for a clause when m = 0 or n = 0.

If n = 0, read

for all $x_1,\ldots,x_k$, B_1 or ... or B_m.

If m = 0, read

for no $x_1, \ldots, x_k$, A_1 and ... and A_n,

or, equivalently, read

for all $x_1, \ldots, x_k$, not A_1 or ... or not A_n.

If n=0 and m=0 the clause is the null clause and it is to be read as a contradiction.

Example

Let CAT be a sentence of resolution logic:

$$\begin{aligned} \text{CAT} = \quad & \{+\text{Cat}(\textit{nil},y,y) \\ & ,+\text{Cat}(\text{cons}(u,x),y,\text{cons}(u,z))\text{-Cat}(x,y,z) \\ & \} \end{aligned}$$

According to the above pragmatics the sentence

$$\text{CAT} \cup \{\text{-Cat}(\text{cons}(a,\textit{nil}),\text{cons}(b,\textit{nil}),\text{cons}(a,\text{cons}(b,\textit{nil})))\} \quad (4.1)$$

is to be read as

for all y, Cat(*nil*,y,y)

and for all u,x,y,z,Cat(cons(u,x),y,cons(u,z)) is implied by

Cat(x,y,z)

and not Cat(cons(*a*,*nil*),cons(*b*,*nil*),cons(*a*,cons(*b*,*nil*))) (4.2)

The sentence CAT can be used to define the catenation relation among linear lists.

Let me first explain how to recognize linear lists here. A linear list I define to be either the empty list or a nonempty list of which the 'head' is an atom and of which the 'tail' is a linear list (possibly empty). Now consider the Herbrand universe of (4.1). Think of the constant *nil* as the empty list and think of the constants *a* and *b* as atoms. Think of a term cons (α_1,α_2) in the Herbrand universe as the linear list of which the head is α_1 and of which the tail is α_2, provided that α_1 is an atom and α_2 is a linear list (possibly *nil*).

In this way the term

cons(*a*,cons(*b*,*nil*))

is to be throught of as the linear list containing the atoms a and b in that order. This is an awfully cumbersome notation. For the benefit of this and all further

examples in the paper I will streamline the notation as follows. Write '.' instead of 'cons', write it in infix notation instead of in prefix notation and adopt the convention that, in the absence of brackets, pairs associate from the right to the left. That is, instead of cons(*a*,cons(*b*,*nil*)), I write *a.b.nil* and this means *a.*(*b.nil*) rather than (*a.b*).*nil*. This notational simplification comes from PROLOG. Using it, the sentence (4.1) becomes

$$\begin{array}{l}\{+\mathrm{Cat}(nil,y,y)\\ ,+\mathrm{Cat}(u,x,y,u.z)\ \text{-}\mathrm{Cat}(x,y,z)\\ ,\text{-}\mathrm{Cat}(a.nil,b.nil,a.b.nil)\\ \}\end{array} \qquad (4.3)$$

In order to appreciate the sentence CAT as a way to define the catenation relation among linear lists, note that *a.b.nil* is the result of catenating *a.nil* and *b.nil* in that order. The sentence (4.3) is unsatisfiable because it denies this fact. This suggests that we can define in general the catenation relation as the set of triples $\alpha_1, \alpha_2, \alpha_3$ of linear lists such that

$$\mathrm{CAT} \cup \{\text{-}\mathrm{Cat}(\alpha_1, \alpha_2, \alpha_3)\}$$

is unsatisfiable.

The problem of sorting stated in resolution logic

$$\begin{array}{l}\mathrm{SORT} = \{+\mathrm{Sort}(x,y)\ \text{-}\mathrm{Perm}(x,y)\ \text{-}\mathrm{Ord}(y)\\ \quad ,+\mathrm{Perm}(nil,nil)\\ \quad ,+\mathrm{Perm}(x.y,z)\ \text{-}\mathrm{Cat}(v1,x.v2,z)\ \text{-}\mathrm{Cat}(v1,v2,v)\ \text{-}\mathrm{Perm}(y,v)\\ \quad ,+\mathrm{Ord}(nil),+\mathrm{Ord}(x.nil)\\ \quad ,+\mathrm{Ord}(x.y.z)\ \text{-}\mathrm{Inf}(x,y)\ \text{-}\mathrm{Ord}(y.z)\\ \quad \}\end{array}$$

The sentence SORT will be used to define the sortedness relation among linear lists. The clauses of SORT are intended to be read as follows:

1) y is the sorted version of x if y is a permutation of x and if y is ordered
2) *nil* is a permutation of *nil*
3) z is a permutation of x.y if v is the result of deleting x from z (see Cat as defined by CAT) and if v is a permutation of y
4) the empty list is ordered
5) any one-element list is ordered
6) a list of at least two atoms is ordered if the first two are in order (see Inf as defined below) and if the tail of that list is ordered

The basic ordering relation between atoms, as expressed in Inf, still has to be defined. Such a definition will have to depend on which atoms there are. For instance, if atoms are letters of the alphabet then the definition can be a

sentence containing a clause for each pair of letters in the order relation:

$$
\begin{aligned}
\mathrm{INF} = \{&+\mathrm{Inf}(a,a), +\mathrm{Inf}(b,b), +\mathrm{Inf}(c,c), \ldots, +\mathrm{Inf}(z,z)\\
&,+\mathrm{Inf}(a,b), +\mathrm{Inf}(b,c), \ldots\\
&,+\mathrm{Inf}(a,c), \ldots\\
&,\ldots \qquad\qquad \ldots\\
&\qquad \ldots \quad ,+\mathrm{Inf}(c,z)\\
&\ldots \quad ,+\mathrm{Inf}(b,z)\\
&,+\mathrm{Inf}(a,z)\\
&\}
\end{aligned}
$$

A part of the data contained in INF can be deduced when the properties of transitivity and reflexitivity are given as in

$$
\begin{aligned}
\mathrm{INF1} = \{&+\mathrm{Inf}(x,x)\ \text{-Letter}(x)\\
&,+\mathrm{Inf}(x,y)\ \text{-P}(x,y)\\
&,+\mathrm{P}(a,b), +\mathrm{P}(b,c), \ldots, +\mathrm{P}(y,z)\\
&,+\mathrm{P}(x,z)\ \text{-P}(x,y)\ \text{-P}(y,z)\\
&\qquad \text{-Letter}(x)\ \text{-Letter}(y)\ \text{-Letter}(z)\\
&,+\mathrm{Letter}(a), \ldots, +\mathrm{Letter}(z)\\
&\}
\end{aligned}
$$

INF1 has fewer clauses but a proof procedure will in general have to go through several steps in order to prove what is required. In INF anything that can be deduced from it is explicitly present. INF1 has the disadvantage that a long deduction may often be necessary, for instance, if a and z have to be compared often. It may be useful in such a case to add redundantly to INF1 the ready-made fact $+\mathrm{Inf}(a,z)$.

Such an arrangement would be reminiscent of Michie's 'memo-functions' (Michie, 1968). Those familiar with this mechanism will see that it can be used to produce hybrids of INF and INF1 that adapt advantageously to a given pattern of usage.

It should be noted that in PROLOG there is no need for the programmer to give a definition of Inf at all: the system calls instead a brief FORTRAN subroutine for the comparison of characters or numbers. Such 'built-in' predicates exist for several other generally useful functions.

The sentence A = SORTUCATUINF defines the relation of sortedness among linear lists in the following way:

$A \cup \{\text{-Sort}(\alpha_1,\alpha_2)\}$ is unsatisfiable if α_1 and α_2 are linear lists and if α_2 is the sorted version of α_1. Moreover,

$$A \cup \{\text{-Sort}(\alpha_1,y)\} \tag{4.4}$$

is unsatisfiable for any linear list α_1 (made up of atoms accounted for in INF). Later we shall see that the sentence (4.4) can cause a proof procedure to con-

struct α_2, a sorted version of α_1. For the moment, when we have not yet discussed proof procedures, this construction is best described semantically.

Because (4.4) is unsatisfiable there exists, by a theorem of Herbrand, an unsatisfiable finite set of variable-free instances of clauses in this sentence. All that need be said now about the proof procedure is that it will construct such a set and that this set contains only one instance of -Sort (α_1,y). The substitution for y in that instance is α_2, a sorted version of α_1.

This is in fact the realization in resolution logic of Green's simulation method of automatic programming (see Figure 3.1). In the simulation method one can made a computer sort lists without anyone ever having to write a program to do it: we only need a program for a complete resolution proof procedure and the specification A.

Is this an interesting alternative to the writing of a sorting program? I believe it is not. I know of no proof procedure that does any better with this specification than to generate a permutation of α_1 until it is found to violate order, then to generate the next, and so on until an ordered permutation is encountered.

Although it will be universally agreed that this application of simulation is a computational disaster, not everybody will agree on the cause. Most work in automatic resolution theorem-proving prior to about 1970 seems to be based on the assumption that the causes of such a disappointing result can be cured by changes to the proof-procedure. The subsequent lack of success caused most workers in automatic problem-solving to discard resolution logic altogether, thus throwing away the baby with the bath water. The more immediately successful *ad hoc* methods advocated by Minsky and Papert (see their 'Uniform Proof Procedures versus Heuristic Knowledge' in (Minsky and Papert, 1972)) carried the day.

In the problem of sorting, however, there is no need to take recourse to such methods: the cause of the disappointing result is in the specification. A good proof procedure is a necessary, not a sufficient, condition for computational success.

There are often different ways to specify the same relation, and there are among equivalent, correct specifications computationally good ones and computationally bad ones (with respect to a given type of proof procedure), just as there are among equivalent, correct programs efficient ones and inefficient ones (with respect to a given machine).

I will exhibit a specification of the sortedness relation for which the simulation method does give efficient computations. The specification will turn out to have a very direct relationship to the 'quicksort' algorithm (Hoare, 1971). In order to make the discovery of the specification as understandable as possible and also to introduce the idea of resolution logic as a program language, I will devote the next section to a transformation of an Algol program for quicksort into a specification in resolution logic for sortedness giving efficient computations according to the simulation method with a generally useful proof procedure.

A specification according to the quicksort principle

Above I have contrasted descriptive against imperative specifications of algorithms. Descriptive specifications are mainly concerned with *what* an algorithm does and they are characteristic of high-level program languages. Imperative specifications are more concerned with *how* an algorithm does whatever it does. Such specifications are characteristic of low-level program languages. A language like Algol 60 is in principle completely imperative: every statement is a command to be executed. Yet sometimes such a program has descriptive value as well as an imperative effect: a well-written program can often be understood by a human reader without explicit comments as to what is to be done by the algorithm.

I start out with an Algol program for the quicksort program written as descriptively as I can. I then eliminate some of the remaining imperative features by writing successive versions in hypothetical generalizations of Algol 60. The end result reads (almost) like a specification in logic of the sortedness relation, which I will then also give. In the next section we will study this specification as a program in resolution logic. This interpretation will be facilitated by the Algol programs of this section.

In the original formulation of the quicksort algorithm (Hoare, 1961) the sequence to be sorted is represented as an array. This makes an efficient program possible. Here it is preferable to disregard questions of efficiency at this level and to choose a data representation that requires a minimal distraction from the algorithm; hence the choice of lists as representation for the sequences to be sorted and for the results of sorting. However, in Algol 60 lists do not exist and I start therefore with a hypothetical extension, 'AlgolX', of Algol 60 that has as data types *atom* and *list*, a constant *nil* and standard one-place functions 'head' and 'tail' applicable to a non-empty list with the usual result, and the two-place function '.' in infix notation, applicable to an atom and a list giving the usual list as result.

```
procedure Sort(x,y);
begin if x≠nil
      then begin atom x1; list x2,u1,u2,v1,v2
            ;     x1:=head(x); x2:=tail(x)
            ;     Part(x1,x2,u1,u2)
            ;     Sort(u1,v1); Sort(u2,v2)
            ;     Cat(v1,x1.v2,y)
            end
;     if x=nil then y:=nil
end
```

FIG. 4.1. Quicksort in AlgolX

In Figure 4.1 the main components are calls to procedures without side effects, passing their results by means of output parameters. A non-empty list x is sorted by decomposing it into the first atom xl (its head) and the remaining list x2 (its tail), possibly empty. 'Part' partitions x2 into two lists ul and u2. The union of the sets of their elements is the set of the elements of x2; ul contains all (if any) of the elements <xl, u2 contains all (if any) of the elements >xl. Of the elements =xl it does not matter in which of ul or u2 they are contained, as long as it is in exactly one of these. The lists ul and u2 are sorted by recursive calls to 'Sort', giving v1 and v2. The sorted version y of x is obtained by catenating (by a call to 'Cat') v1 and v2 with x1 in between.

The identity x=head(x).tail(x) for any non-empty list x suggests that the selectors 'head' and 'tail' are superfluous in the presence of the constructor '.'. Consider for instance the declaration of 'Sort' as given in Figure 4.2.

```
procedure Sort(x,y);
begin atom x1; list x2,u1,u2,v1,v2
;     if x=x1.x2
      then begin  Part(x1,x2,u1,u2)
                ;     Sort(u1,v1); Sort(u2,v2)
                ;     Cat(v1,x1.v2,y)
                end
;     if x=nil then y:=nil
;
end
```

FIG. 4.2. Quicksort in AlgolY (AlgolX plus pattern matching)

For the program in Figure 4.2 the usual Algol interpreter will give an error message when attempting to evaluate the Boolean expression x=x1.x2 because the variables x1 and x2 are without value. Suppose, however, that the interpreter would not be neutral with respect to truth and falsity but would prefer true expressions to false ones. Then it might notice that even though the expression is not actually true, it can be made true without violating any existing values: the as yet undefined variables x1 and x2 can be assigned the values head(x) and tail(x) respectively and then the Boolean expression is true. We then say that x has been 'matched' to x1.x2.

But this can only be done if x≠nil. So why not combine the explicit assignation and the test x≠nil into one matching operation? The advantage is that in this way a low-level, machine-oriented operation like assignation is avoided. Note that matching is a generalization of equality in Algol 60 because when there are no undefined variables, matching reduces to the usual evaluation of Boolean

equalities in Algol 60. Matching is known as a feature of several program languages for AI research (Bobrow and Raphael, 1974).

Note that in the program in Figure 4.2 the input parameter x seems to be quite unnecessary for communicating the algorithm. If we want to know whether the actual parameter for x is of the form xl.x2, why not have that form itself as a formal parameter? This suggests having two procedure declarations for 'Sort', applicable according to whether the input list is empty. Another extension of Algol is required, say AlgolZ. The proposal is to extend AlgolY to allow more than one procedure call with the same name. A declaration responds to a call when the actual parameters of the call match the formal parameters of the declaration. If the match is successful, the name is replaced by the body where undefined variables have received the values necessitated by the match. If more than one declaration is applicable, it is not determined which is actually applied: the program is indeterminate. This proposal follows the PLANNER language for AI research where the feature is called "procedure invocation by pattern matching".

```
procedure Sort(x1.x2,y);
begin list u1,u2,v1,v2
;     Part(x1,x2,u1,u2)
;     Sort(u1,v1); Sort(u2,v2)
;     Cat(v1,x1.v2,y)
end
; procedure Sort(nil,nil)
;
```

FIG. 4.3. Quicksort in AlgolZ

I stated that the program in Figure 4.1 was as descriptive as I could make it in AlgolX. This is not quite true: the program requires Sort(u1,v1) to be executed before Sort(u2,v2). This is an example of what E.W. Dijkstra calls "sequential overspecification" because in fact the order is irrelevant and should therefore remain unspecified. All three programs in Figures 4.1, 4.2, and 4.3 suffer from this defect, which is remedied in Figure 4.4. Note the disappointing property of Algol that one sometimes has to write more when one wants to specify less.

In Figure 4.4 I suppose that Part and Cat have become Boolean procedures as well; with the same effect on their parameters as before, but assuming the Boolean value **true** if their execution completes successfully.

Note that the effect of the body of Sort(x1.x2,y) in Figure 4.4 is not necessarily the same as when it would have been

Sort:=
Part(x1,x2,u1,u2)∧Sort(u1,v1)∧Sort(u2,v2)∧Cat(v1,xl.v2,y)

```
Boolean procedure Sort(x1.x2,y);
begin list u1,u2,v1,v2; Boolean success
;     success:=Part(x1,x2,u1,u2)
;     success:=success∧Sort(u1,v1)∧Sort(u2,v2)
;     Sort:=success∧Cat(v1,x1.v2,y)
end
; Boolean procedure Sort(nil,nil); Sort:=true
; .....
```

FIG. 4.4. A better Quicksort in AlgolZ

This body does not specify any order between the calls, at least not according to the definition (Naur, 1963) of Algol 60. This body would be an example of sequential underspecification: I would have gone too far in eliminating imperative features. Note that a declaration with this body can be read as:

Sort(xl.x2,y) is true if
Part(x1,x2,u1,u2), Sort(u1,v1), Sort(u2,v2), and Cat(v1,x1.v2,y) are true

This suggests as specification of sorting in first-order predicate logic:

$$(\forall x1,x2,y.\ (\exists u1,u2,v1,v2.\ \mathrm{Part}(x1,x2,u1,u2)\wedge\mathrm{Sort}(u1,v1)\wedge\mathrm{Sort}(u2,v2)\wedge\mathrm{Cat}(v1,x1.v2,y)) \supset \mathrm{Sort}(x1.x2,y))\wedge\mathrm{Sort}(\mathit{nil},\mathit{nil})$$

The same sentence in resolution logic, together with specifications of Part and Cat can now be given as follows:

```
SORT1 = {+Sort(x1.x2,y)  -Part(x1,x2,u1,u2)
                         -Sort(u1,v1) -Sort(u2,v2)
                         -Cat(v1,x1.v2,y)
        ,+Sort(nil,nil)
        ,+Part(x1,z.x2,z.u1,u2) -Inf(z,x1)   -Part(x1,x2,u1,u2)
        ,+Part(x1,z.x2,u1,z.u2) -Inf(x1,z)   -Part(x1,x2,u1,u2)
        ,+Part(x,nil,nil,nil)
        ,+Cat(u.x,y,u.z) -Cat(x,y,z)
        ,+Cat(nil,y,y)
        }
```

In a previous section I exhibited a sentence SORT as specification for a sorting algorithm giving a very disappointing behaviour when used for automatic programming in Green's simulation mode. I claimed that the cause was in the

specification rather than inherent in the method. SORT1 is a specification of the sortedness relation in the same sense:

$$\text{INF} \cup \text{SORT1} \cup \{\text{-Sort}(\alpha_1, y)\}$$

is unsatisfiable for any linear list α_1 made up of atoms for which the order is specified in INF. A complete resolution procedure will construct as substitution for y the sorted version of α_1 in the course of proving unsatisfiability. In the next section I will discuss Kowalski's proof procedure that will perform the construction with the same efficiency as an AlgolZ interpreter would execute the program in Figure 4.3.

KOWALSKI'S PROCEDURAL INTERPRETATION OF RESOLUTION LOGIC

The LUSH system of inference rule and proof procedure

A clause containing one positive literal is called a *regular clause*. A sentence containing only regular clauses is called a *regular sentence*. A clause containing no positive literal is called a *goal statement*. A goal statement containing no negative literal is called a *halt statement* and is written, as before, as □.

Up till now we have met with a syntax, a semantics, and a pragmatics for resolution logic. The procedural interpretation can be regarded as a supplement to the pragmatics given before. For specifications of relations we have learned the importance of unsatisfiability of a sentence. It is now time to consider a procedure for constructing a proof of unsatisfiability for an unsatisfiable sentence. For this we will only consider sentences of a special form: containing, apart from one goal statement, only regular clauses.

There are two interrelated reasons for dealing only with sentences with regular clauses and one goal statement. The first is that the LUSH rule of inference and proof procedure are designed for such sentences. The other reason is that for regular clauses and for goal statments there exists a pragmatics complementary to the one discussed in 4.3, namely the relevant part of Kowalski's procedural interpretation of resolution logic. As can be learned from (Kowalski, 1974c), the procedural interpretation is not restricted to regular clauses and goal statements.

The LUSH system is due to Kowalski (Kowalski, 1974a; 1974c). In these papers the system has not received a name. The name LUSH was coined by Hill (Hill, 1974), the excuse being given as: 'Linear resolution with Unrestricted Selection for Horn clauses'. Clauses containing at most one literal are usually called Horn clauses honouring the pioneering investigation (Horn, 1951) of A. Horn of some of their properties. I prefer to follow A. Colmerauer (unpublished work) and use the term regular clause, as defined above, to contain exactly one positive literal.

The LUSH rule of inference infers a new goal statement

$$(\text{-}A_1 \ldots \text{-}A_{i-1} \text{-}B_1 \ldots \text{-}B_m \text{-}A_{i+1} \ldots \text{-}A_n)\theta$$

from a goal statement

$$-A_1 \ldots -A_{i-1}-A_i-A_{i+1} \ldots -A_n$$

with $-A_i$ as *selected literal* and a regular clause (if one exists)

$$+A\ -B_1 \ldots -B_m$$

that matches the goal statement in the sense that there exists a "most general" substitution θ of terms for variables (see (Nilsson, 1971)) that makes A_i and A identical. The LUSH rule of inference is a resolution with a goal statement and a regular clause as parents.

The proof procedure of LUSH determines how the rule of inference is used to construct a proof. One component of it is the *selection rule*; the other component is the *search strategy*. The rule of inference and the selection rule determine together with a sentence $S \cup \{G\}$ (S a regular sentence, G a goal statement) a tree of (which the nodes are) goal statements (also called the *search space*) in the following way. The root of the tree is G. The descendants of a node N in the tree are determined by first applying the selection rule to N, thus obtaining a literal L. There is just one descendant of N for each different way in which the rule of inference can be applied to a regular clause in S and the goal statement N with L as selected literal.

A path in the tree from the root to a halt statement represents a proof of the unsatisfiability of $S \cup \{G\}$. Suppose the path is the sequence of goal statements $G = G_0, G_1, \ldots, G_n = \square$. The rule of inference is such that, for $i = n-1, \ldots 0$, if $S \cup \{G_{i+1}\}$ is unsatisfiable then so is $S \cup \{G_i\}$. $S \cup \{\square\}$ is unsatisfiable, therefore $S \cup \{G\}$ is.

For theoretical investigations an important property of LUSH is its completeness (Hill, 1974) which holds when S is regular and G is a goal statement: if $S \cup \{G\}$ is unsatisfiable, then the tree must contain a halt statement, whatever the selection rule. Completeness in this sense obviously does not mean that any path from the root ends in a halt statement. It does not preclude the possibility of infinite paths in the tree or of finite paths not ending in a halt statement. Completeness is sometimes asserted of a combination of search space and search strategy meaning that the search space always contains a halt statement and that the search strategy always finds a path to it.

Predicate-logic programming as discussed in this paper is restricted to application of LUSH to proving the unsatisfiability of a sentence $S \cup \{G\}$, S a regular sentence, G a goal statement. Neither Kowalski's procedural interpretation of predicate-logic nor the PROLOG system is thus restricted. Yet LUSH represents, as it were, the backbone of the PROLOG system. Because PROLOG has been shown to be a program language of considerable heuristic power, it is interesting to know what are the possible selection rules and what are the possible search strategies in that part of PROLOG that corresponds to LUSH.

The selection rule of PROLOG is the one that always selects the leftmost literal. Only those search spaces can therefore be implemented which can be

obtained by ordering, once and for all in a given program, the negative literals of a clause. The search strategy is a depth-first, leftmost-descendant-first search of the tree determined by the selection rule just given. To further define the search strategy an ordering has to be given for the descendants. The ordering is determined by the ordering of the regular clauses in S: if clauses C_1 and C_2 occur in that order and both match the selected literal in a goal statement G then the descendant of G, obtained by matching with C_1, is generated first. Again, note that the search strategy is fixed once and for all by the order of the clauses in S.

The procedural interpretation

In order to use resolution logic for goal-directed computations another pragmatics is useful in addition to the one discussed before. According to Kowalski's procedural interpretation (Kowalski, 1974a) a regular clause

$$+A\ -B_1\ \ldots\ -B_m \qquad\qquad m = 0,1,\ldots$$

is interpreted as a *procedure definition*. The positive literal +A is interpreted as the *procedure name*. The negative literals $-B_1,\ldots,-B_m$ are interpreted as *procedure calls* constituting the *procedure body* of the definition. The body may be empty (when m=0); such a procedure definition is interpreted as an *assertion of fact*.

A goal statement, that is a clause containing no positive literal, is interpreted as a set of procedure calls to be executed. When a goal statement is empty there are no more procedures to be executed; it is therefore called a *halt statement*.

The procedural interpretation is based on an analogy between the inference rule of LUSH and the computation rule for procedure-oriented program languages that executes a procedure call by replacing the call by the body of a declaration of which the name matches the call. This is just what happens in an application of the LUSH rule of inference as described in the previous section: $-A_i$ is the procedure call selected for execution from the goal statement, A is the name of a matching declaration, and the body replacing the call $-A_i$ is $-B_1,\ldots\ -B_m$. The substitution θ modifies the body in a way that corresponds to replacing formal by actual parameters. The fact that the other procedure calls in the original goal statement may also be modified by θ as a result of executing $-A_i$ is an interesting generalization of the usual computation rule for procedure-oriented languages.

Suppose the selection rule of LUSH is the one that always selects the leftmost literal of the goal statement for execution. Then the goal statement acts like the stack of procedures called but not yet executed as used in a typical implementation of the computation rule for procedure-oriented languages. The leftmost literal corresponds to the top of the stack. Other selection rules correspond to departures from the stack-disciplined implementation, such as, for instance, the use of co-routines.

It may be verified that LUSH (with the selection rule that always selects the leftmost literal) acting on the sentence:

$$\text{SORT1} \cup \{\text{-Sort}(\alpha_1, y)\}$$

produces the sorted version of α_1 with approximately the same number of procedure calls as an Algol Z interpreter would require for sorting α_1 with the program in Figure 4.3. This comparison assumes that Part and Cat would also be programmed in AlgolZ using procedure calls only. For this to be achieved it is necessary that the tree of goal statements contains essentially only one path and, of course, that this path terminates and that it terminates in the halt statement. A selection rule that causes the tree to have this form is the one that always selects the leftmost literal with the literals ordered as in the listing of SORT.

Example

The list *c.a.b.nil* may be sorted by demonstrating the unsatisfiability of the sentence SORT1 ∪ INF ∪ {-Sort (*c.a.b.nil*,y)}. Given that the selection rule always selects the leftmost literal, an initial part of the search space is:

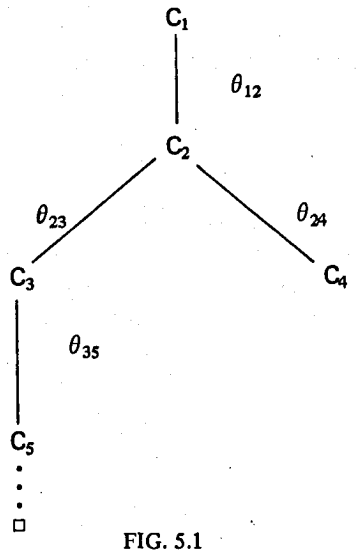


FIG. 5.1

In Figure 5.1 there are the following goal statements:

$C_1 = \{\text{-Sort}(c.a.b.nil, y)\}$
$C_2 = \{\text{-Part}(c.a.b.nil, p1, p2)\} \cup C$
$C_3 = \{\text{-Inf}(a,c)\ \text{-Part}(c, b.nil, u1, u2)\} \cup C\,\theta_{23}$
$C_4 = \{\text{-Inf}(c,a)\ \text{-Part}(c, b.nil, u1, u2)\} \cup C\,\theta_{24}$
$C_5 = \{\text{-Part}(c, b.nil, u1, u2)\} \cup C\,\theta_{23}\,\theta_{35}$

where

$C = \{\text{-Sort}(p1,q1)\ \text{-Sort}(p2,q2)\ \text{-Cat}(q1,c.q2,y)\}$

In Figure 5.1 there are the following substitutions:

θ_{12} = (x1,x2:=*c,a.b.nil*)
θ_{23} = (x1,z,x2,p1,p2:=*c,a,b.nil,a*.u1,u2)
θ_{24} = (x1,z,x2,p1,p2:=*c,a,b.nil*,u1,*a*.u2)
θ_{35} = identity

In Figure 5.1 the tree of goal statements has more than one path. Wherever a branch occurs (when Part is the predicate of the selected literal), it has two arms, one of which ends at the next goal statement. Notice the important role played by the selection rule. Suppose that in the goal statement C_4 any literal would have been selected other than -Inf(*c,a*), the leftmost. Then the goal statement would have had at least one descendant, and that one perhaps again, and so on, whereas the presence of -Inf(*c,a*) makes it impossible for the halt statement to be an ultimate descendant. The selection -Inf(*c,a*) ensures that the fact, that the wrong turn has been taken at the branch, is detected before time is wasted on other goals.

That this happens depends on the fact that the search space for INF $\cup$ $\{$-Inf(*c,a*)$\}$ is finite. A terminating search reports the absence of a halt statement. The search space for INF1 $\cup$ $\{$-Inf(*c,a*)$\}$ is infinite and does not contain a halt statement either. A search for a halt statement will not terminate. Therefore a LUSH proof procedure selecting the leftmost literal and using a depth-first search strategy will not in general succeed in sorting by proving the unsatisfiability of

$$\text{SORT1} \cup \text{INF1} \cup \{\text{-Sort}(\alpha_1, y)\}$$

where α_1 is a linear list containing only atoms occurring in INF1. This in spite of the fact that INF and INF1 are equivalent in the sense that

$$\text{INF} \cup \{\text{-Inf}(\alpha_1, \alpha_2)\}$$

is unsatisfiable if and only if

$$\text{INF1} \cup \{\text{-Inf}(\alpha_1, \alpha_2)\}$$

is unsatisfiable, for any variable-free terms α_1 and α_2.

What happens at branches in the tree of Figure 5.1 illustrates how "conditionals" can be handled in predicate-logic programming. Where an Algol-like language would have a conditional statement, a specification in logic will have two declarations with the same name, say +P. In general, with a selected literal -P, the goal statement has two descendants, one for each declaration with name +P. Sometimes the call matches only one name, as in the goal statement C_1 of Figure 5.1: procedure call by pattern matching has carried out the test whether the input list is empty. Sometimes the call matches both names, as in the goal statement C_2. The test is then deferred one step because in the descendant goal

statements the selected literals represent mutually disjoint contingencies. One of the descendants must remain without issue. When both descendants continue on to a halt statement, we have the analogue of an indeterminate algorithm.

The specification SORT1 is not only a specification for quicksort, because, for instance, the sentence

$$\text{SORT1} \cup \text{INF} \cup \{\text{-Sort(x,}a.b.c.nil\text{)}\}$$

is unsatisfiable. LUSH is able to prove it is, and will substitute for x any of the lists $a.b.c.nil, a.c.b.nil, b.a.c.nil, b.c.a.nil, c.a.b.nil$, and $c.b.a.nil$ of which the sorted version is $a.b.c.nil$. The specification SORT1 can apparently also be used as a permutation algorithm, provided that LUSH used a different selection rule.

Let us see what happens if the selection rule is not changed, that is, if the leftmost literal is always selected, which worked so well for sorting. The tree of goal statements would begin as shown in Figure 5.2.

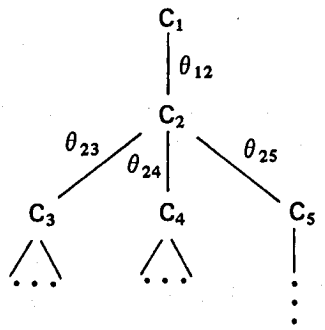


FIG. 5.2

In Figure 5.2 the clauses are

$$C_1 = \{\text{-Sort(x,}a.b.c.nil\text{)}\}$$
$$C_2 = \{\text{-Part(x1,w2,p1,p2)}\} \cup C$$
$$C_3 = \{\text{-Inf(z,x1) -Part(x1,x2,u1,u2)}\} \cup C\theta_{23}$$
$$C_4 = \{\text{-Inf(x1,z) -Part(x1,x2,u1,u2)}\} \cup C\theta_{24}$$
$$C_5 = \{\text{-Sort(}nil\text{,v1) -Sort(}nil\text{,v2) -Cat(v1,x.v2,}a.b.c.nil\text{)}\}$$

where

$$C = \{\text{-Sort(p1,v1) -Sort(p2,v2) -Cat(v1,x1.v2,}a.b.c.nil\text{)}\}$$

In Figure 5.2 the substitutions are

$$\theta_{12} = \text{(x,y:=x1.w2,}a.b.c.nil\text{)}$$
$$\theta_{23} = \text{(w2,p1,p2:=z.x2,z.u1,u2)}$$
$$\theta_{24} = \text{(w2,p1,p2:=z.x2,u1,z.u2)}$$
$$\theta_{25} = \text{(x1,w2,p1,p2:=x,}nil\text{,}nil\text{,}nil\text{)}$$

We now have a very large search space. C_5 has one descendant clause only, but no halt statement as ultimate descendant. C_3 and C_4 have hundreds of immediate descendants each. Rather than to try and follow in detail the search space, let us try to get an overview of what happens in it. It will be useful to extend the notion of "descendant" to literals, as follows. A literal L_2 in a clause C_2 is a descendant of a literal L_1 in a clause C_1 if C_2 is a descendant of C_1 and L_1 is the selected literal in C_1 and if L_2 is in the procedure body that replaces L_1 in C_1 to give C_2. Also if L_3 is a descendant of L_2 and if L_2 is a descendant of L_1 then L_3 is a descendant of L_1.

When the leftmost literal is always selected, descendants of -Part(x1,w2, p1,u2) will be selected as long as there are any. By the time there are no more such descendants, x1,w2,p1,u2 will have been substituted by any terms that happen to fit in the partition relation. The input list *a.b.c.nil* has not been taken into account at all. We see that with a wrong selection rule LUSH behaves as "blindly" as has often been observed with earlier, faulty applications of resolution theorem-proving.

A good selection rule should select for execution a procedure call containing input data, in this case

-Cat(v1,x1.v2,*a.b.c.nil*)

The outputs of this procedure call can act as input to the calls to Sort and after finishing those there are data for executing Part. With a selection rule that imposes this ordering on calls, the tree of goal statements is quite different: it may be verified that in that case all substantial branches lead to success; a different success each time: each path to a halt statement builds a substitution that gives the x in the original goal statement a different permutation as value. We have obtained an adequate permutation generator. We have just seen an example of the possibility in predicate-logic programming, pointed out by Kowalski (Kowalski, 1974a), of computing different functions from the same relational specification by using different combinations of arguments for input and output.

The selection rule required to achieve successful use as permutation generator is again one that can easily be specified in PROLOG by ordering the literals in such a way that the PROLOG rule of selecting the leftmost literal executes the right one every time. A correct ordering is achieved by interchanging the literals -Part(x1,x2,u1,u2) and -Cat(v1,xl.v2,y) in SORT1. The interchange has been carried out in PERM, shown below, where also a clause has been added which causes the permutations to be printed. The predicate Print can be defined in such a way that in this case all permutations are printed rather than just the first one found.

As a result of having PROLOG execute the sentence

$$\text{PERM} \cup \text{INF} \cup \{\text{-Perm}(a.b.c.nil)\}$$

all permutations of the list *a.b.c.nil* are printed out.

```
PERM =   {+Sort(x1.x2,y)   -Cat(v1,x1.v2,y)
                           -Sort(u1,v1) -Sort(u2,v2)
                           -Part(x1,x2,u1,u2)
         ,+Sort(nil,nil)
         ,+Part(x1,z.x2,z.u1,u2) -Inf(z,x1) -Part(x1,x2,u1,u2)
         ,+Part(x1,z.x2,u1,z.u2) -Inf(x1,z) -Part(x1,x2,u1,u2)
         ,+Part(x,nil,nil,nil)
         ,+Cat(u.x,y,u.z) -Cat (x,y,z)
         ,+Cat(nil,y,y)
         ,+Perm(y) -Sort(x,y) -Print(x)
         }
```

If one would just be interested in obtaining a permutation generator, INF and the calls -Inf(. . .) would be deleted. They only restrict the program as a permutation generator by making it necessary to give the list in sorted order. But here I want to keep the connection with sorting as simple as possible.

PREDICATE-LOGIC PROGRAMS FOR SYNTACTICAL ANALYSIS

Left-recursivity, search space, and search strategy

I will first explain the method of Colmerauer and Kowalski (Colmerauer, 1973; Kowalski, 1974c) for representing strings and grammars in predicate logic. Strings are represented by lists. Whenever it has to be asserted that a substring is adjacent to another, the substrings are represented by 'differences' of lists. Two lists x and y are defined to have a difference if there exists a list z such that x is the result of catenating z and y, in that order. The difference is the string of the atoms in z. For instance, *a.b.c.nil* and *c.nil* have a difference which is the string of the atoms *a* and *b*.

Let us consider a formal grammar $G = (V_N, V_T, P, S)$ where V_N is the set of nonterminals, V_T the set of terminals, P the set of productions and S the start symbol. Take for example a context-free left-recursive grammar for 'unsigned integers'

```
G = (VN = {Ui,D}
    ,VT = {0,1,2,3,4,5,6,7,8,9}
    ,P  = {Ui→D
           ,Ui→Ui D
           ,D→0,D→1,D→2,D→3,D→4
           ,D→5,D→6,D→7,D→8,D→9
           }
    ,S  = Ui
    )
```

According to the method of Colmerauer and Kowalski the production rules are represented by the following sentence of resolution logic:

$$
\begin{aligned}
UIL = \{&+Ui(x,y)\ -D(x,y)\\
&,+Ui(x,z)\ -Ui(x,y)\ -D(y,z)\\
&,+D(0.y,y),+D(1.y,y),+D(2.y,y),+D(3.y,y),+D(4.y,y)\\
&,+D(5.y,y),+D(6.y,y),+D(7.y,y),+D(8.y,y),+D(9.y,y)\\
&\}
\end{aligned}
$$

The formula +Ui(x,z) asserts that the substring represented by the difference of lists x and z is of the syntactic category Ui. The second clause reads: the difference between x and z is a Ui if there exists a y such that the difference between x and y is a Ui and the difference between y and z is a D.

The grammar G is left-recursive. It is interesting to see how this fact is reflected in the search space of LUSH selecting the leftmost literal for the sentence

$$UIL \cup \{-Ui(3.9.2.nil,nil)\}$$

In showing this sentence to be unsatisfiable, LUSH recognizes in effect the difference between *3.9.2.nil* and *nil* as an unsigned integer.

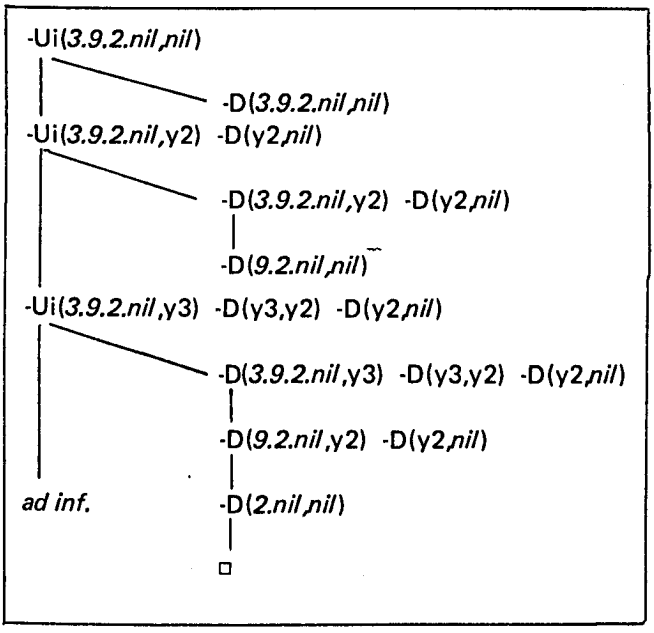


FIG. 6.1. Search space for a left-recursive grammar

The combination of left-recursivity and the rule selecting the leftmost literal for the LUSH rule of inference results in the search space of Figure 6.1 which has an infinite branch. Whether LUSH will find the parse depends on the search strategy. For instance, PROLOG's depth-first search strategy will find the parse

when the first two clauses are ordered as shown in UIL, but not when they would have been ordered in the other way.

The right-recursive version of the grammar UIL is represented by the following sentence:

$$\begin{aligned}\mathrm{UIR} = \{&+\mathrm{Ui}(x,y)\,-\mathrm{D}(x,y)\\ &,+\mathrm{Ui}(x,z)\,-\mathrm{D}(x,y)\,-\mathrm{Ui}(y,z)\\ &,+\mathrm{D}(0.y,y),+\mathrm{D}(1.y,y),+\mathrm{D}(2.y,y),+\mathrm{D}(3.y,y),+\mathrm{D}(4.y,y)\\ &,+\mathrm{D}(5.y,y),+\mathrm{D}(6.y,y),+\mathrm{D}(7.y,y),+\mathrm{D}(8.y,y),+\mathrm{D}(9.y,y)\\ &\}\end{aligned}$$

In Figure 6.2 we find the search space for LUSH selecting the leftmost literal for the sentence

$$\mathrm{UIR} \cup \{-\mathrm{Ui}(3.9.2.nil,nil)\}$$

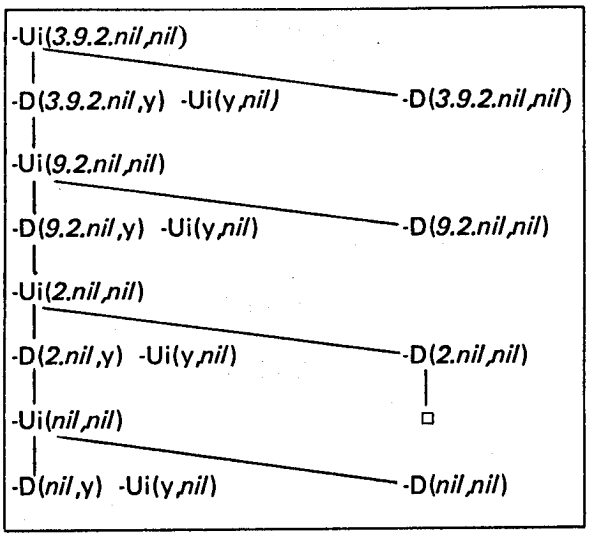


FIG. 6.2. Search space for a right-recursive grammar

Here the search space is finite: any search strategy will find the parse. In PROLOG one can handle the left-recursive grammar, but only by choosing the right ordering of clauses. The right-recursive grammar is easier in the sense that ordering of clauses does not matter a great deal.

A. Colmerauer (unpublished work) has given a formal definition of formal grammars and of the parsing problem in resolution logic according to the method of Colmerauer and Kowalski and has proved that there exists a LUSH derivation if and only if a parse exists. The parse can be reconstructed from the successive resolutions performed during the parse. A 'safe' search strategy, like breadth-first search of the tree of goal statements is guaranteed to find a parse if one exists; this cannot be said in general of PROLOG.

The regular sentences of resolution logic as generalized grammars

For the syntactical analysis of natural language context-free grammars are not suitable. For example, the necessity to express agreement in number (singular and plural) between a noun phrase and the corresponding verb phrase gives rise to an unmanageable proliferation of productions. The problem is discussed by Winograd (Winograd, 1972), who goes on to argue that context-sensitive grammars, although an improvement in this respect, are not satisfactory either. He prefers to define a language not by a grammar, but as the set of strings successfully analysed by a program in a program language named PROGRAMMAR.

I do not think one should leave it at this: besides having a program, it is also useful to have a machine-independent description of the language and an easy method to convince oneself of the correctness of the program, that is, that the program indeed performs according to the description. According to the method of Colmerauer and Kowalski one can describe in resolution logic a language in a machine-independent way. With respect to a given (suitable) proof procedure. the machine-independent description becomes a (useful) program.

I will illustrate some of the distinctive features of predicate-logic programming by an application to syntactic analysis and especially by comparing it to the PROGRAMMAR approach. In this section I will give descriptions in predicate-logic (that can be run as PROLOG programs) based on two of Winograd's example PROGRAMMAR programs, namely, his 'Grammar 2 and 'Grammar 3'.

In his first example, Winograd gives both an explicit machine-independent description as a context-free grammar ('Grammar 1') and the corresponding PROGRAMMAR program ('Grammar 2'). To obtain a PROLOG program, all we have to do is to transcribe Grammar 1 systematically according to the representation of Colmerauer and Kowalski for strings and productions. The transcription is in fact so systematic that PROLOG has a facility, 'SuperQ', that allows direct input of grammars in a notation similar to the one of Colmerauer's Q-systems (Colmerauer, 1973). But for just the two examples here it is not worth introducing a new notation. The remaining examples are therefore also in the language of resolution logic as defined in this paper.

The grammar G1 includes Winograd's Grammar 1:

```
G1 = (V_N={Sent,Np,Vp,Det,Noun,Iverb,Tverb}
     ,V_T={the,giraffe,apple,dreams,eats}
     ,P  ={Sent→Np Vp
           ,Np→Det Noun
           ,Vp→Iverb,Vp→Tverb Np
           ,Det→the,Noun→giraffe,Noun→apple
           ,Iverb→dreams,Tverb→dreams,Tverb→eats
           }
     ,S  = Sent
     )
```

According to the method of Colmerauer and Kowalski the productions of this grammar are represented by the following sentence of resolution logic:

G2 = {+Sent(x,y) -Np(x,u) -Vp(u,y)
,+Np(x,y) -Det(x,u) -Noun(u,y)
,+Vp(x,y) -Iverb(x,y)
,+Vp(x,y) -Tverb(x,u) -Np(u,y)
,+Det(*the*.y,y),+Noun(*giraffe*.y,y)
,+Noun(*apple*.y,y),+Iverb(*dreams*.y,y)
,+Tverb(*dreams*.y,y),+Tverb(*eats*.y,y)
}

In order to analyze "the giraffe dreams" LUSH is set to work on the sentence

$$G2 \cup \{\text{-Sent}(\textit{the.giraffe.dreams.nil,nil})\}$$

With the rule selecting leftmost literals we find the search space of Figure 6.3.

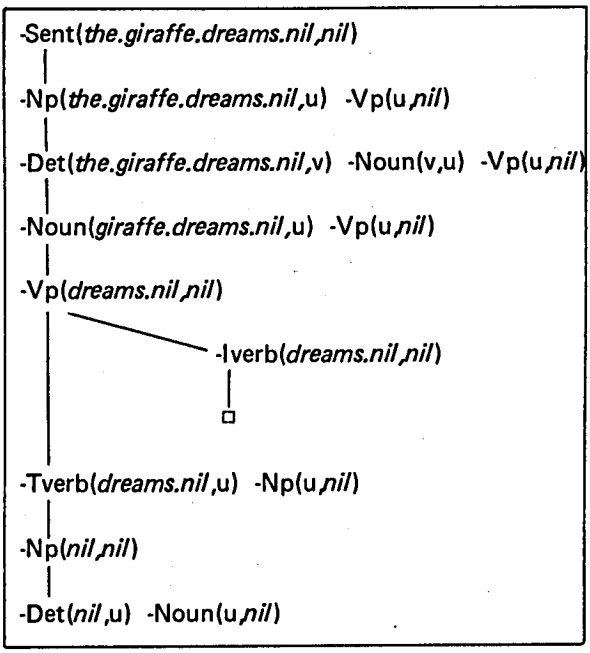


FIG. 6.3. The search space for 'the giraffe dreams'

The sentence G2 is a direct representation in predicate-logic of the facts expressed in G1. These facts are what has been called above the logic component of an algorithm for analysing sentences produced by G1. What has been called the control component determines, for instance, that a parser is top-down rather than bottom-up. When the control component is provided by a LUSH proof procedure, the resulting algorithm will be a top-down parser. In the PROGRAMMAR program 'Grammar 2' logic and control are not clearly separated.

The advantage of separating the logic and control components is that programs become more understandable, easier to get right, and to modify. Perhaps a useful way to make more precise the distinction between 'descriptive' versus 'imperative,' or 'high-level' versus 'low-level' algorithm specification is to identify both distinctions with whether the logic and control components are separated.

When the logic and control components are separated, it can be recognized easily when two algorithms are the same except for the control component. For example, in the previous section we saw almost identical algorithms for quicksort and for a permutation generator differing only in their control components. For example, two parsers, one top-down and another bottom-up, for the same formal language should, according to the principle of separating logic from control, have a clearly recognizable and easily readable part in common, namely some machine-independent representation of a grammar for the language. This is achieved in predicate-logic programming, but not in PROGRAMMAR.

For parsers, the separation of logic and control is not new. It has been achieved in compiler-generators, which contain a program that takes a grammar as input to 'compile' it into a program that parses according to that grammar. LUSH takes both a grammar and a string (combined in one unsatisfiable sentence) as input to produce a parse, working in 'interpreter' rather than in 'compiler' mode. Another difference is that the program in a compiler-generator that produces a parser will not do anything else. PROLOG not only accepts predicate-logic programs specifying any context-free grammar (and certain non-restrictive classes of type-0 and type-1 grammars as well, within the framework of method of Colmerauer and Kowalski) but also programs for anything else that fits the paradigm of predicate-logic programming. LUSH may be regarded as a generalized top-down parser and the regular sentences of resolution logic may be regarded as generalized context-free grammars which turn out to be convenient for programming a great variety of problems, like the sorting and permuting algorithms shown in this paper, or the problems arising in the large application programs mentioned earlier (Bergman and Kanoui, 1973; Colmerauer, 1974; Colmerauer, *et al.*, 1972; Kanoui, 1973; Paséro, 1973; Warren, 1974).

The sentence G2 can be interpreted as a logic component conveying no control information. In that case the selection rule and the search strategy will have to be stated explicitly when using LUSH. Or one can make use of the fact that one cannot avoid writing down an unordered set of literals or of clauses in some order. In PROLOG a LUSH proof procedure interprets the order of the literals as a selection rule and the order of the clauses as a particular choice of depth-first search strategy. PROLOG makes it possible to provide explicit control information by means of two system-defined predicates. It may be surprising to find how useful such limited possibilities for supplying control information can be.

As far as I can see, there is nothing in resolution logic that makes it more difficult to supply control information than in, say, PLANNER-like languages. There is no need to supply this information *in* logic or to camouflage it *as* logic,

as is done in PROLOG. A more elaborate control language, as used in MICRO-PLANNER or CONNIVER, could be applied to direct LUSH, or other proof-procedures, which can combine top-down and bottom-up deductions, or those which, like the connection-graph theorem-prover (Kowalski, 1974b), transcend the distinction between top-down and bottom-up.

Let us now continue with Winograd's examples and see how context-sensitive aspects can be introduced in his Grammar 1. I will use the language of the regular clauses of resolution logic as a generalized grammar of great expressive power, rather than try to fit into the form grammars of the Chomsky hierarchy. The advantage of separating logic from control will become much more apparent here, because in PROGRAMMAR the context-sensitive aspects are handled by explicit instructions for moving a pointer about in the parse tree as it exists at the moment of execution of the instruction. In order to be able to understand Winograd's Grammar 3 in PROGRAMMAR one has to have a mental picture of the parse tree, and the position of the pointer, as it changes during execution.

It is therefore not easy to discover what the grammar is in Grammar 3. Following Winograd's comments, I assume it is Grammar 1 with the following elaborations:

1) the number of a noun phrase agrees with that of the corresponding verb phrase
2) the number of a noun phrase need not be determined by the noun only (this fish – these fish) and not by the 'th-word' only (the giraffe – the giraffes), but number is a feature of the entire noun phrase
3) a noun phrase need not have a determiner (giraffes dream), presumably provided by the noun phrase is plural

G3 = {+Sent(x,y) -Np(n,x,u) -Vp(n,u,y)
,+Np(n,x,y) -Th(n,x,u) -Noun(n,u,y)
,+Np(*pl*,x,y) -Noun(*pl*,x,y)
,+Vp(n,x,y) -Verb(*i*,n,x,y)
,+Vp(n,x,y) -Verb(*t*,n,x,u) -Np(m,u,y)
}

INTERFACE = {+Noun(*sg*,u.y,y) -Npr(u,v)
,+Noun(*pl*,v.y,y) -Npr(u,v)
,+Verb(tty,*sg*,u.y,y) -Conj(tty,u,v)
,+Verb(tty,*pl*,v.y,y) -Conj(tty,u,v)
}

LEXICON = {+Npr(*giraffe,giraffes*),+Npr(*fish,fish*),
,+Npr(*dream,dreams*)
,+Conj(tty,*dreams,dream*),+Conj(tty,*eats,eat*)
,+Th(n,*the*.y,y)
,+Th(*sg,this*.y,y),+Th(*pl*,*these*.y,y)
}

Differences between G2 and G3 are the following. The predicates have received extra parameters for transmitting information about number (variables n and m, taking as possible values the constants *sg* for singular or *pl* for plural) or about whether or not a verb is transitive (variable tty, taking as possible values the constants *i* or *t*). The last two parameters of literals in G3 are, as before, lists indicating where the substring under consideration begins and ends (variables u, x, and y). Lexical information has been removed from G3.

Lexical information has been placed in a separate sentence, LEXICON, giving pairs (Npr) with the singular and plural form of some nouns. At a further level of refinement the description will have to analyse the structure of the words themselves, for instance, to be able to say that, apart from exceptions, the plural is formed by appending an s. As it stands, the words in LEXICON are constants and have therefore no internal structure. Similarly, verbs are given by simply listing (with Conj) various conjugations; in the example these are restricted to the singular and plural of the third person, present, indicative. Finally, singular and plural forms of the 'th-words' *the* and *this* are listed. Many, quite different, *ad hoc* or more systematic, representations of the lexicon are usable with the same grammar G3, if a suitable interface is provided.

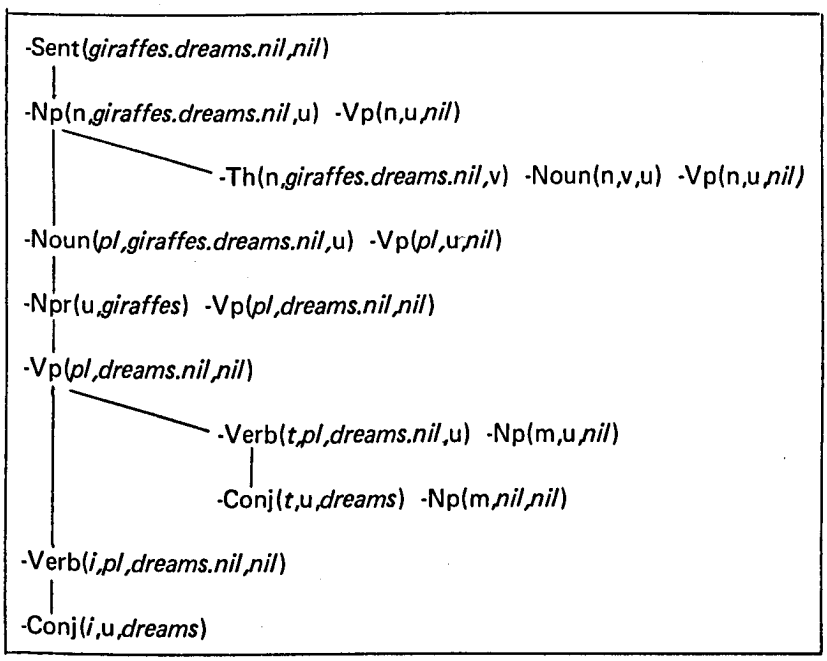


FIG. 6.4. The search space for 'giraffes dreams'

As the final example, I show (Figure 6.4) the search space for LUSH with the rule selecting the leftmost literal for the unsolvable parsing problem represented by the satisfiable sentence

G3 ∪ INTERFACE ∪ LEXICON ∪ {-Sent(*giraffes.dreams.nil,nil*)}

The search space is finite; LUSH will terminate, whatever the search strategy, without having found a halt statement.

ACKNOWLEDGMENTS

This paper grew out of an earlier attempt ('First-order predicate logic as a high-level program language', Research Memorandum MIP-R-106, Dept. of Machine Intelligence, University of Edinburgh, 1974) at a tutorial exposition of programming with resolution logic. The discussions with Robert Kowalski and the encouragement of Donald Michie have helped me very much with the writing of MIP-R-106, which was supported by the U.K. Science Research Council. Additional support was provided by a research grant from the University of Waterloo.

REFERENCES

Annual review in automatic programming. (1960-1970) Pergamon Press.

Battani, G. and Meloni, H. (1973) Interpreteur du language de programmation PROLOG. *Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille.*

Bergman, M. and Kanoui, H. (1973) Application of mechanical theorem to symbolic calculus. *Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille.*

Bledsoe, W.W. (1971) Splitting and reduction heuristics in automatic theorem-proving *Artificial Intelligence* **2**, 55-77.

Bledsoe, W.W., Boyer, R. and Henneman, W.H. (1972) Computer proofs of limit theorems. *Artificial Intelligence* **3**, 27-60.

Bobrow, D. and Raphael, B. (1974) New programming languages for AI research. *Computing Surveys* **6**, 153-174.

Colmerauer, A. (1973) Les systèmes-Q, ou un formalisme pour analyser et synthetiser des phrases sur ordinateur. *Publication interne no 43, Dépt. d'Informatique, Faculté des Sciences, Université de Montréal.*

Colmerauer, A. (1974) Programmation en langue naturelle. *Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille.*

Colmerauer, A., Kanoui, H., Paséro, R. and Roussel, P. (1972) Un système de communication homme-machine en français. *Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille.*

Elcock, E.W. (1968) Descriptions. *Machine Intelligence* **3** (ed. D. Michie) Edinburgh University Press, Edinburgh, pp. 173-179.

van Emden, M.H. and Kowalski, R.A. (1976) The semantics of predicate logic as a programming language, *JACM* **23**, to appear.

Foster, J.M. (1968) Assertions: programs written without specifying unnecessary order. *Machine Intelligence* **3**, (eds. Meltzer, B. and Michie, D.) Edinburgh University Press, Edinburgh, pp. 387-392.

Foster, J.M. and Elcock, E.W. (1969) Absys 1: an incremental compiler for assertions. *Machine Intelligence* **4**, (eds. Meltzer, B. and Michie, D.) Edinburgh University Press, Edinburgh, pp. 423-439.

Green, C.C. (1969) The application of theorem-proving to question-answering systems. *Technical Note 8*, Artificial Intelligence Group, Stanford Research Institute.

Hayes, P.J. (1973) Computation and deduction. *Proc. 1973 MFCS Conference*, Czechoslovakian Academy of Sciences.

Hill, R. (1974) LUSH resolution and its completeness. *DCL Memo 78*, Dept. of Artificial

Intelligence, University of Edinburgh.
Hoare, C.A.R. (1961) Algorithm 64: quicksort. *CACM* 4, 321.
Horn, A. (1951) On sentences which are true of direct unions of algebras. *J. Symbolic Logic*, **16**, 14-21.
Kanoui, H. (1973) Application de la demonstration automatique aux manipulations algébriques et à l'integration formelle sur ordinateur. *Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille.*
Kleene, S.C. (1961) *Mathematical Logic.* Wiley.
Kowalski, R.A. (1974a) Predicate logic as programming language, *Proc. IFIP74*, North Holland, pp. 569-574.
Kowalski, R.A. (1974b) A proof procedure using connection graphs. *DCL Memo 74*, University of Edinburgh.
Kowalski, R.A. (1974c) Logic for problem-solving. *DCL Memo 75*, Dept. of Artificial Intelligence, University of Edinburgh.
Kowalski, R.A. (1975) *Inaugural Lecture.* Imperial College, London.
Kowalski, R.A. and Kuehner, D. (1971) Linear resolution with selection function. *Artificial Intelligence* **2**, 227-260.
Loveland, D.W. (1969) A simplified format for the model-elimination theorem-proving procedure. *JACM* **16**, 349-363.
Michie, D. (1968) Memo functions and machine learning. *Nature* **218**, 19-22.
Michie, D., Ross, R. and Shannan, G.J. (1972) G-deduction. *Machine Intelligence* **7**, (eds. Meltzer, B. and Michie, D.), Edinburgh University Press, Edinburgh, pp. 141,165.
Michie, D. and Sibert, E. (1974) Some binary derivation systems. *JACM* **21**, 175-190.
Minsky, M. and Papert, S. (1972) Progress Report. *Memo 252*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.
Naur, P. (ed.) (1963) Revised Report on the Algorithmic Language Algol 60. *CACM* **6**, 1-17; *Comp. J.* **5**, 349-367; *Numer. Math.* **4**, 420-453.
Nilsson, N.J. (1971) *Problem-solving methods in artificial intelligence.* McGraw-Hill.
Paséro, R. (1973) Representation du français en logique du premier ordre en vue de dialoguer avec un ordinateur. *Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille.*
Reiter, R. (1968) Two results on ordering for resolution with merging and linear format. *JACM* **15**, 630-646.
Robinson, J.A. (1965) A machine-oriented logic based on the resolution princple. *JACM* **12**, 23-44.
Robinson, J.A. (1967) Heuristic and complete processes in the mechanization of theorem-proving. *Systems and Computer Science*, (eds. Hart, J.F. and Takasu, S.), University of Toronto Press.
Warren, D. (1974) WARPLAN: a system for generating plans. *DCL Memo 76*, Dept. of Artificial Intelligence, University of Edinburgh.
Winograd, T. (1972) *Understanding Natural Language*, Academic Press and Edinburgh University Press.

16

An Overview of Production Systems

Randall Davis and Jonathan King

Computer Science Department
Stanford University

INTRODUCTION AND BACKGROUND

Since production systems (PS) were first proposed by Post (Post, 1943) as a general computational mechanism, the methodology has seen a great deal of development and has been applied to a diverse collection of problems. Despite the wide scope of goals and perspectives demonstrated by the various systems, there appear to be many recurrent themes. This paper is an attempt to provide an analysis and overview of those themes, as well as a conceptual framework by which many of the seemingly disparate efforts can be viewed, both in relation to each other, and to other methodologies[1].

Accordingly, we use the term 'production system' in a broad sense, and attempt to show how most systems which have used the term can be fit into the framework. The comparison to other methodologies is intended to provide a view of PS characteristics in a broader context, with primary reference to procedurally-based techniques, but with reference also to some of the current developments in programming and the organization of data and knowledge bases.

We begin by offering a review of the essential structure and function of a PS, presenting a picture of a "pure" PS to provide a basis for subsequent elaborations. We then suggest that current views of PSs fall into two distinct classes, and demonstrate that this dichotomy may explain much of the existing variation in goals and methods. This is followed by some speculations on the nature of appropriate and inappropriate problem domains for PSs–i.e., what is it about a problem that makes the PS methodology appropriate, and how do these factors arise out of the system's basic structure and function? Next we review a dozen different characteristics which we found common to all systems, explaining how they contribute to the basic character, and noting their interrelationships. Finally, we present a taxonomy for PSs, selecting four dimensions of characterization, and indicating the range of possibilities as suggested by recent efforts.

Two points of methodology should be noted–first, we make frequent

reference to what is "typically" found, and what is "in the spirit of things." Since there is really no one formal design for current PSs, and recent implementations have explored variations on virtually every aspect, their use becomes more an issue of a programming *style* than anything else. It is difficult, then, to exclude designs or methods on formal grounds, and we refer instead to an informal, but, we feel, well established style of approach.

A second, related point will be important to keep in mind as we compare the capabilities of PSs with those of other approaches. Since it is possible to imagine coding any given Turing machine in either procedural or PS terms (see [Anderson, 1976] for a formal proof of the latter), in the formal sense their computational power is equivalent. This suggests that, given sufficient effort, they are ultimately capable of solving the same problems. The issues we wish to examine are not, however, questions of absolute computational power, but the impact of a particular methodology on program structure, as well as the relative ease or difficulty with which certain capabilities can be achieved, and the extent to which they can be achieved "in the spirit of things."

"PURE" PRODUCTION SYSTEMS

A production system may be viewed as consisting of three basic components: a set of rules, a data base, and an interpreter for the rules. In the simplest design, a rule is an ordered pair of symbol strings, with a left and right hand side (LHS and RHS); the rule set has a predetermined, total ordering; and the data base is simply a collection of symbols. The interpreter in this simple design operates by scanning the LHS of each rule until one is found which can be successfully matched against the data base. At that point the symbols matched in the data base are replaced with those found in the RHS of the rule, and scanning either continues with the next rule or begins again with the first. A rule can also be viewed as simple conditional statement, and the invocation of rules as a chained sequence of *modus ponens* actions.

Rules

More generally, one side of a rule is *evaluated* with reference to the data base, and if this succeeds (i.e., evaluates to TRUE in some sense), the action specified by the other side is performed. Note that *evaluate* is typically taken to mean a passive operation of "perception" or "an operation involving only matching and detection,"[2] while the action is generally one or more conceptually primitive operations. As noted, the simplest evaluation is a matching of literals, and the simplest action is a replacement. Some of the variations on this include the use of "match and refresh" as the evaluation process (the matched symbols are relocated in the data base as a result of being matched), or the use of "addition" as the action process (rather than being replaced, the matched symbols remain in the data base and the new symbols are simply added to the collection). More complex constructs are also being examined; see "Programmability." As we

explore further below, the choice of one or another of these operations is generally based on the nature of the task domain.

Note that we do not specify which side is to be matched, since either is possible. For example, given a grammar written in production rule form[3],

$$S \rightarrow A\ B\ A$$

$$B \rightarrow B\ 0 \qquad A \rightarrow A\ 1$$
$$B \rightarrow 0 \qquad A \rightarrow 1$$

matching the LHS on a data base which consists of the start symbol S, gives a generator for strings in the language. Matching on the RHS of the same set of rules gives a recognizer for the language. We can also vary the methodology slightly to obtain a top down recognizer, by interpreting elements of the LHS as goals to be obtained by the successful matching of elements from the RHS. In this case the rules "unwind." Thus we can use the same set of rules in several ways. Note, however, that in doing so we obtain quite different systems, with characteristically different control structures and behavior.

The organization and accessing of the rule set is also an important issue. The simplest scheme is the fixed, total ordering mentioned, but elaborations quickly grow more complex. The term *conflict resolution* has been used to describe the process of selecting a rule. These issues of rule evaluation and organization are explored in more detail below.

Data base

In the simplest production system, the data base is simply a collection of symbols intended to reflect the state of the world, but the interpretation of those symbols depends in large part on the nature of the application. For those systems intended to explore symbol processing aspects of human cognition, the data base is interpreted as modelling the contents of some memory mechanism (typically short term memory, STM), with each symbol representing some "chunk" of knowledge, and hence its total length (typically around seven elements), and organization (linear, hierarchical, etc.), are important theoretical issues. Typical contents of STM for psychological models are those of PSG (Newell, 1973), where STM might contain purely content-free symbols like:

```
QQ
(EE FF)
TT
```

or VIS (Moran, 1973a), where the STM contains symbols representing directions on a visualized map:

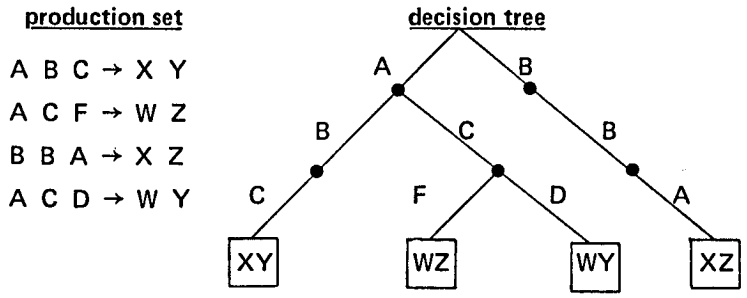
```
(NEW C-1 CORNER WEST L-1 NORTH L-2)
(L-2 LINE EAST P-2 P-1)
(HEAR NORTH EAST % END)
```

For systems intended to be knowledge-based experts, the data base contains facts and assertions about the world, is typically of arbitrary size, and has no *a priori* constraints on the complexity of organization. For example, the MYCIN system (Davis, 1975) uses a collection of 4-tuples, consisting of an associative triple and a "certainty factor" (CF, see [Shortliffe, 1975b]), which indicates (on a scale from -1 to 1) how strongly the fact has been confirmed (CF > 0) or disconfirmed (CF < 0):

```
(IDENTITY ORGANISM-1 E.COLI.8)
(SITE CULTURE-2 BLOOD 1.0)
(SENSITIVE ORGANISM-1 PENICILLIN -1.0)
```

As another example, in the DENDRAL system (Feigenbaum, 1971; Smith, 1972) the data base contains complex graph structures which represent molecules and molecular fragments. The structures are built by assigning unique numbers to each atom of a molecule and by describing chemical bonds by a pair of numbers indicating the atoms they join.

A third style of organization for the data base is the "token stream" approach used, for example, in LISP70 (Tesler, 1973). Here the data base is a linear stream of tokens, accessible only in sequence. Each production in turn is matched against the beginning of the stream (i.e., if the first character of a production and the first character of the stream differ, the whole match fails), and if the rule is invoked, it may act to add, delete, or modify characters in the matched segment. The anchoring of the match at the first token offers the possibility of great efficiency in rule selection, since the productions can be "compiled" into a decision tree which keys off sequential tokens from the stream (a very simple example is shown below).



Whatever the organization of the data base, one important characteristic should be noted: it is the sole storage medium for all state variables of the system. In particular, unlike procedurally-oriented languages there is no provision for separate storage of control state information—no separate program counter, pushdown stack, etc. There is nothing but the single data base, and all information to be recorded must go there. We refer to this as the unity of data and control store, and examine some of its implications below. This store is,

moreover, universally accessible to every rule in the system, so that anything put there is potentially detectable by any rule. We will see that both of these have significant consequences for the use of the data base as a communcation channel.

Interpreter

The interpreter is the source of much of the variation found among different systems, but it may be seen in the simplest terms as a *select-execute* loop, in which one rule applicable to the current state of the data base is chosen, and then executed. Its action results in a modified data base, and the select phase begins again. Given that the selection is often a process of choosing the first rule which matches the current data base, it is clear why this cycle is often referred to as a 'recognize-act', or 'situation-action' loop. The range of variations on this theme is explored in the section on control cycle architecture.

This alternation of selection and execution is an essential element of PS architecture that is responsible for one of their most fundamental characteristics. By choosing each new rule for execution on the basis of the total contents of the data base, we are effectively performing a complete reevaluation of the control state of the system at every cycle (recall the unity of data and control store). This is distinctly different from procedurally-oriented approaches, in which control flow is typically the decision of the process currently executing, and is commonly dependent on only a small fraction of the total number of state variables. PSs are thus sensitive to any change in the entire environment, and potentially reactive to such changes with the scope of a single execution cycle. The price of such reactivity is, of course, the computation time required for the reevaluation.

An example of one execution of the recognize-act loop for a much simplified version of Newell's PSG system will illustrate some of the foregoing notions. The production system, called PS.ONE, is assumed for this example to contain two productions, PD1 and PD2. We indicate this as follows:

PS.ONE:(PD1 PD2)

PD1:(DD and (EE) → BB)
PD2:(XX → CC DD)

PD1 says that if the symbol DD and some expression beginning with EE (that is, (EE . . .)) is found in STM, then insert the symbol BB at the front of STM. PD2 says that if the symbol XX is found in STM, then first insert the symbol CC, then the symbol DD, at the front of the STM. The initial contents of STM are:

STM: (QQ (EE FF) RR XX SS)

This STM is assumed to have a fixed maximum capacity of five elements. As new elements are inserted at the beginning (left) of the STM, therefore, other elements will be lost (forgotten) off the right end. Second, as illustrated below, for

this system elements accessed in matching the condition of a rule are 'refreshed' (pulled to the front of STM), rather than replaced.

The production system then scans the productions in order: PD1, then PD2. Only PD2 matches, so it is evoked. The contents of STM after this step are:

STM: (DD CC XX QQ (EE FF))

PD1 will match during the upcoming cycle to yield,

STM: (BB DD (EE FF) CC XX)

completing two cycles of the system.

TWO VIEWS OF PRODUCTION SYSTEMS

Throughout much of the work reported, there appear to be two major views of PSs, as characterized on one hand by the psychological modelling efforts (PSG, PAS II, VIS, etc.), and on the other by the performance oriented, knowledge-based expert systems (e.g., MYCIN, DENDRAL). These appear to be two distinct efforts which have arrived at similar methodologies while pursuing differing goals.

The efforts to simulate human performance on simple tasks are aimed at creation of a program which embodies a theory of that behavior. From the performance record of experimental subjects the modeller attempts to formulate the minimally competent set of production rules, the smallest set still able to reproduce the behavior. Note that 'behavior' here is meant to include *all* aspects of human performance (including mistakes, the effects of forgetting, etc.), all shortcomings or successes which may arise out of (and hence may be clues to) the 'architecture' of human cognitive systems[4].

An example of this approach is the PSG system, from which we constructed the example above. This system has been used to test a number of theories to explain the results of the Sternberg memory scanning tasks (Newell, 1975), with each set of productions representing a different theory of how the human subject retains and recalls the information given to him during the psychological task. Here the subject first memorizes a small subset of a class of familiar symbols (e.g., digits), and then attempts to respond to a symbol flashed on a screen by indicating whether or not it was in the initial set. His response times are noted.

The task was first simulated with a simple production system that performed correctly, but did not account for timing variations (which were due to list length and other factors). Refinements were then developed to incorporate new hypotheses about how the symbols were brought into memory, and eventually a good simulation was built around a small number of productions.

Newell has reported (Newell, 1973) that use of a production system methodology led in this case to a novel hypothesis that certain timing effects are caused by a decoding process rather than a search process, and that it also clearly

illustrated the possible tradeoffs in speed and accuracy between differing processing strategies. As such it was an effective vehicle for the expression and evaluation of theories of behavior.

The performance-oriented expert systems, on the other hand, start with productions as a representation of knowledge about a task or domain, and attempt to build a program which displays competent behavior in that domain. These efforts are not concerned with similarities between the resulting systems and human performance (except insofar as the latter may provide a possible hint about ways to structure the domain or to approach the problem, or as a yardstick for success, since few AI programs approach human levels of competence). They are intended simply to perform the task without errors of any sort, human-like or otherwise.

The approach is characterized by the DENDRAL system, in which much of the development has involved embedding a chemist's knowledge about mass spectrometry into rules usable by the program, without attempting to model the chemist's thinking. The program's knowledge is extended by adding rules that apply to new classes of chemical compounds. Similarly, much of the work on the MYCIN system has involved crystallizing informal knowledge of clinical medicine in a set of production rules.

Despite the difference in emphasis, both approaches have been drawn to PSs as a methodology. For the psychological modellers, production rules offer a clear, formal, and powerful way of expressing basic symbol processing acts, which form the primitives of information processing psychology[5]. For the designer of knowledge-based systems, production rules offer a representation of knowledge that is relatively easily accessed and modified, making it quite useful for systems designed for incremental approaches to competence. For example, much of the MYCIN system's capability for explaining its actions is based on the representation of knowledge as individual production rules (Shortliffe, 1975a). This makes the knowledge far more accessable to the program itself than it might otherwise be if it were embodied in the form of ALGOL-like procedures. As in DENDRAL, the modification and upgrading of the system are by incremental modification of, or addition to the rule set.

Note that we are suggesting here, and through much of the rest of the paper, that it is possible to view a great deal of the work on PSs in terms of a unifying formalism. The intent is to offer a conceptual structure which can help organize what may appear to be a disparate collection of efforts. The presence of such a formalism should not, however, obscure the significant differences which arise out of the various perspectives. As one example, the decision to use RHS-driven rules in a goal-directed fashion implies a control structure which is simple and direct, but relatively inflexible. This offers a very different programming tool than the LHS-driven systems. The latter are capable of much more complex control structures, giving them capabilities much closer to those of a complete programming language. Recent efforts, especially, have begun to explore the issues of more complex, higher level control within the PS methodology (see

"Programmability," below).

It should also be noted that production systems are seen by some (Newell, 1972) not as simply a convenient paradigm for approaching psychological modelling, but rather as a methodology whose power arises out of its close similarity to fundamental mechanisms of human cognition. In this view, human problem solving behavior can be modelled easily and successfully by a production system because it in fact is being generated by one:

> We confess to a strong premonition that the actual organization of human programs closely resembles the production system organization.... We cannot yet prove the correctness of this judgment, and we suspect that the ultimate verification may depend on this organization's proving relatively satisfactory in many different small ways, no one of them decisive.
>
> In summary, we do not think a conclusive case can be made yet for production systems as *the* appropriate form of [human] program organization. Many of the arguments ... raise difficulties. Nevertheless, our judgment stands that we should choose production systems as the preferred language for expressing programs and program organization.
>
> [Newell, 1972, p. 803-4, 806]

This has led to speculation[6] that the interest in production systems on the part of those building high performance knowledge-based systems is more than a coincidence. It is suggested that this is a result of current research (re)discovering what has been learned by naturally intelligent systems through evolution—that structuring knowledge in a production system format is an effective approach to the organization, retrieval, and use of very large amounts of knowledge.

The success of some production rule based AI systems does give weight to this argument, and the PS methodology is clearly powerful. But whether this is a result of its equivalence to human cognitive processes, and whether this implies artificially intelligent systems ought to be similarly structured, are, we feel, still open questions.

APPROPRIATE AND INAPPROPRIATE DOMAINS

Program designers have found that PSs easily model problems in some domains, but are awkward for others. Let us briefly investigate why this may be so, and relate it to the basic structure and function of a PS.

We can imagine two very different domains—the first is best viewed and understood as consisting of many independent states, while the second seems best understood via a concise, unified theory, perhaps embodied in a single law. Examples of the former include some views of perceptual psychology or clinical medicine, in which there are a large number of states relative to the number of

actions (this may be due either to our lack of a cohesive theory, or to the basic complexity of the system being modelled). Examples of the latter include well-established areas of physics and mathematics, in which a few basic tenets serve to embody much of the required knowledge, and in which the discovery of unifying principles has emphasized the similarities in seemingly different states. This first distinction appears to be one important factor in distinguishing appropriate from inappropriate domains.

A second distinction concerns the complexity of control flow. As two extremes, we can imagine two processes, one of which is a set of independent actions, and the other a complex collection of multiple, parallel processes involving several dependent sub-processes.

A third distinction concerns the extent to which the knowledge to be embedded in a system can be separated from the manner in which it is to be used (also known as the controversy between declarative and procedural representations; see [Winograd, 1975] for an extensive discussion). As one example, we can imagine simply stating facts, perhaps in a language like predicate calculus. This makes no assumptions about the way those facts will be employed. Alternatively, we could write procedural descriptions of how to accomplish a stated goal. Here the use of the knowledge is for the most part predetermined during the process of embodying it in this representation.

In all three of these distinctions, a PS is well suited to the first description and ill suited to the latter. The existence of multiple, nontrivially different, independent states is an indication of the feasibility of writing multiple, non-trivial, modular rules. A process composed of a set of independent actions requires only limited communication between the actions, and, as we shall see, this is an important characteristic of PSs. The ability to state what knowledge ought to be in the system without also describing its use makes an important difference in the ease with which a PS can be written (see "Programmability" below).

For the latter description (unified theory, complex control flow, predetermined use for the knowledge), the economy of the basic theory makes for either trivial rules, or multiple, almost redundant rules. In addition, a complex looping and branching process requires explicit communication between actions, in which one explicitly invokes the next, while interacting subgoals require a similarly advanced communication process to avoid conflict. Such communication is not easily supplied in a PS-based system. The same difficulty also makes it hard to specify in advance exactly how a given fact should be used.

It seems also to be the nature of production systems to focus upon the variations within a domain rather than upon the common threads that link different facts or operations. Thus for example, (as we shall describe in more detail below) the process of addition is naturally expressed via productions as n^2 rewrite operations involving two symbols (the digits being added). The fact that addition is commutative, or rather, that there is a property of "commutativity" shared by all operations that we consider to be addition, is a rather awkward one

to express in production system terms.

This same characteristic may, conversely, be viewed as a capability for focussing on and handling significant amounts of detail. Thus, where the emphasis of a task is on recognition of large numbers of distinct states, PSs provide a significant advantage. In a procedurally-oriented approach, it is both difficult to organize and troublesome to update the repeated checking of large numbers of state variables and the corresponding transfer of control. The task is far easier in PS terms, where each rule can be viewed as a demon awaiting the occurrence of a specific state. [In the case of one current PS (DENDRAL), the initial, procedural approach proved sufficiently inflexible that the entire system was rewritten in production rule terms.]

In addition, we noted above the potential sensitivity and reactivity of PSs which arises from their continual reevaluation of the control state. This has also been referred to as the 'openness' of production rule based systems. It is characterized by the principle that 'any rule can fire at any time', which emphasizes the fact that at any point in the computation, any rule could possibly be the next to be selected, depending only on the state of the data base at the end of the current cycle. Compare this to the normal situation in a procedurally-oriented language, where such a principle is manifestly untrue: it is simply not typically the case that, depending on the contents of that data base, any procedure in the entire program could potentially be the next to be invoked.

PSs therefore appear to be useful where it is important to detect and deal with a large number of independent states, in a system which requires a broad scope of attention, and the capability of reacting quickly to small changes. In addition, where knowledge of the problem domain falls naturally into a sequence of independent 'recognize-act' pairs, PSs offer a convenient formalism for structuring and expressing that knowledge.

Finally, note that the implication is not that both approaches *couldn't* perform in both domains, but that there are tasks for which one of them would prove awkward, and the resulting system unenlightening. Such tasks are far more elegantly accomplished in only one of the two methodologies. The main point is that we can, to some extent, formalize our intuitive notion of which approach seems more appropriate by considering two essential characteristics of any PS–its set of multiple, independent rules, and its limited, indirect channel of interaction via the data base.

PRODUCTION SYSTEM CHARACTERISTICS

Despite the range of variations in methodologies, there appear to be many characteristics common to almost all PSs. It is the presence of these, and their interactions, that contribute to the 'nature' of a PS, its capabilities, deficiencies, and characteristic behaviour. This section is an attempt to isolate and analyze each of these factors, discovering the primitive elements which compose them, and providing an overview of their interactions.

The network of Figure 1 is a summary of features and relationships. Each box

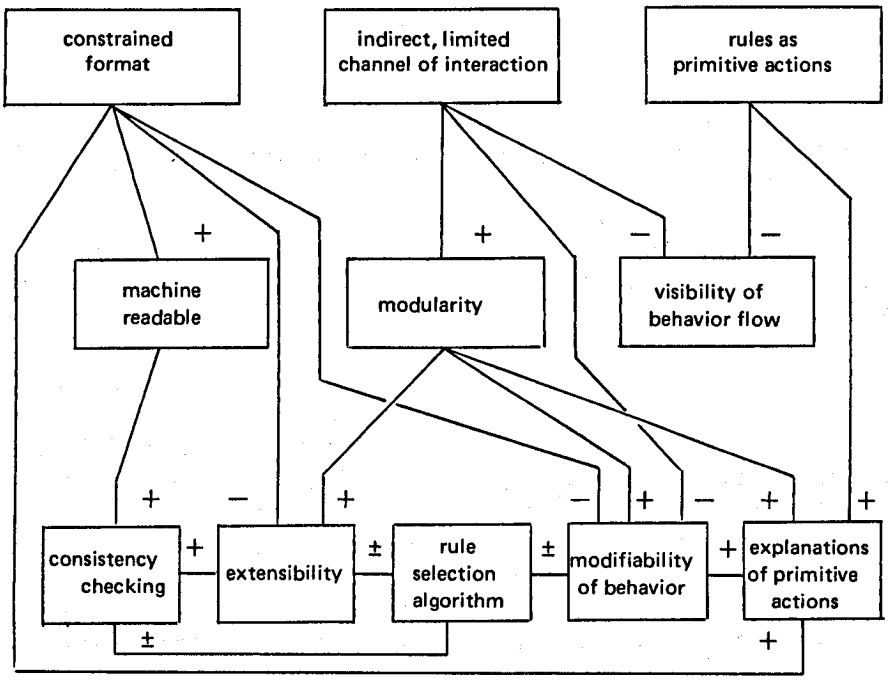


FIG. 1. + indicates facilitation; – indicates inhibition.

represents some feature, capability, or parameter of interest, with arrows labelled with +'s and -'s suggesting the interactions between them. This rough scale of facilitation and inhibition is naturally very crude, but does indicate the interactions as we saw them.

Figure 1 contains at least three conceptually different sorts of factors: (a) those fundamental characteristics of the basic PS scheme (e.g., indirect/limited channel, constrained format); (b) secondary effects of these (e.g., automated modification of behavior); and (c) performance parameters of implementation which are helpful in characterizing PS strengths and weaknesses (e.g., visibility of behavior flow, extensibility). This division of factors is suggested by the three levels indicated in the figure.

Below we briefly describe each feature and suggest the nature of its interactions with the others.

Indirect/limited channel of interaction

Perhaps the most fundamental and significant characteristic of PSs is their restriction on the interactions between rules. In the simplest model, a pure PS, we have a completely ordered set of rules, with no interaction channel other than the data base. The total effect of any rule is determined by its modifications to the data base, and hence subsequent rules must 'read' there any traces it may leave behind. Winograd (Winograd, 1975) characterizes this in discussing global modularity in programming:

> We can view production systems as a programming language in which all interaction is forced through a very narrow channel .... The temporal interactions [of individual productions] is completely determined by the data in this STM, and a uniform ordering regime for deciding which productions will be activated in cases where more than one might apply .... Of course it is possible to use the STM to pass arbitrarily complex messages which embody any degree of interaction we want. But the spirit of the venture is very much opposed to this, and the formalism is interesting to the degree that complex processes can be described without resort to such kludgery, maintaining the clear modularity between the pieces of knowledge and the global process which uses them.

While this characterization is clearly true for a pure PS, with its limitations on the size of STM, we can generalize on it slightly to deal with a broader class of systems. First, in the more general case, the channel is not so much narrow as *indirect* and *unique*. Second, the kludgery arises not from arbitrarily complex messages, but from *specially crafted* messages which force highly specific, carefully chosen interactions.

With reference to the first point, one of the most fundamental characteristics of the pure PS organization is that rules must interact indirectly through a single channel. Indirection implies that all interaction must occur by the effect of modifications written in the data base; uniqueness of the channel implies that these modifications are accessible to every one of the rules. Thus, to produce a system with a specified behavior, one must think not in the usual terms of having one section of code call another explicitly, but rather use an indirect approach in which each piece of code (i.e., each rule) leaves behind the proper traces to trigger the next relevant piece. The uniform access to the channel, along with the openness of PSs, implies that those traces must be constructed in the light of a potential response from any rule in the system. It is in some sense like a difficult case of programming purely by side effects.

With reference to the second point, in many systems the action of a single rule may, quite legitimately, result in the addition of very complex structures to the data base (e.g., DENDRAL, see "Taxonomy" below). Yet another rule in the same system may deposit just one carefully selected symbol, chosen solely because it will serve as an unmistakable symbol to precisely one other (carefully preselected) rule. Choosing the symbol carefully provides a way of sending what becomes a private message through a public channel; the continual reevaluation of the control state assures that the message can take immediate effect. The result is that one rule has effectively called another, procedure style, and this is the variety of kludgery which is contrary to the style of knowledge organization typically associated with a PS. We argue below (see "Visibility,") that it is, in particular, the premeditated nature of such message passing (typically in an attempt to 'produce a system with specified behavior') which is the primary violation of the spirit of PS methodology.

The primary effect of this indirect, limited interaction is to produce a system which is strongly modular, since no rule is ever called directly. It is also, however, perhaps the most significant factor in making the behavior flow of a PS more difficult to analyze. This is due to the fact that, even for very simple tasks, overall behavior of a PS may not be at all evident from a simple review of its rules.

To illustrate many of these issues, consider the algorithm for addition of positive, single digit integers used by Waterman (Waterman, 1974) with his PAS II production system interpreter. First, the procedural version of the algorithm, in which transfer of control is direct and simple:

```
        add(m,n) ::=
A]              count ← 0; nn ← n;
B]      L1:     if count = m then return(nn);
C]              count ← successor(count);
D]              nn ← successor(nn);
E]              go(L1);
```

Compare this with the set of productions for the same task, in Figure 2 (the notation may be somewhat unfamiliar; we have provided a brief explanation). The two are not precisely analogous, since the procedural version does simple addition, while the production set both adds and "learns" (see note [7] and the comments in Figure 2). The example is still quite illustrative, however, and Waterman points out some direct correspondence between productions and statements in the procedure. For example, productions 1 and 2 accomplish the initialization of line A, rule 3 corresponds to line B, and rule 4 to lines C and D. There is no production equivalent to the goto of line E because the production system execution cycle takes care of that implicitly. On the other hand, note that in the procedure, there is no question whatsoever that the initialization step nn ← n is the second statement of "add," and that it is to be executed just once, at the beginning of the procedure. In the productions, the same action is predicated on an unintuitive condition of the STM (essentially it says if the value of n is known, but nn has never been referenced or incremented, then initialize nn to the value that n has at that time). This degree of explicitness is necessary because the production system has no notion that the initialization step has already been performed in the given ordering of statements, so the system must check the conditions each time it goes through a new cycle.

Thus, procedural languages are oriented toward the explicit handling of control flow and stress the importance of its influence on the fundamental organization of the program (as, for example, in recent developments in structured programming). PSs, on the other hand, emphasize the statement of independent chunks of knowledge from a domain, and make a control flow a secondary issue. Given, moreover, the limited form of communication available, it is more difficult to express concepts which require structures larger than a single rule. Thus, where the emphasis is on global behavior of a system rather

than the expression of small chunks of knowledge, PSs are, in general, going to be less transparent than formulations of equivalent routines in procedural terms.

CONDITION	ACTION
1] (READY) (ORDER X1)	⇒ (REP (READY) (COUNT) X1)) (ATTEND)
2] (N X 1) – (NN) – (S NN)	⇒ (DEP (NN X1))
3] (COUNT X1) (M X1) (NN X2) (N X3)	⇒ (SAY X2 IS THE ANSWER) (COND (M X1) (N X3)) (ACTION (STOP)) (ACTION (SAY X2 IS THE ANSWER) (PROD) (STOP)
4] (COUNT) (NN)	⇒ (REP (COUNT) (S COUNT)) (REP (NN) (S NN))
5] (ORDER X1 X2)	⇒ (REP (X1 X2) (X2)) (COND (S X3 X1)) (ACTION (REP (S X3 X1) (X3 X2))) (PROD)

initial STM: (READY) (ORDER 0 1 2 3 4 5 6 7 8 9)

notation:

The Xi's in the CONDITION are variables in the pattern match, all other symbols are literals. An Xi appearing *only* in the ACTION is also taken as a literal. Thus if rule 5 is matched with X1 = 4 and X2 = 5, as its second action it would deposit (COND(S X3 4)) in STM.

All elements of the LHS must be matched for a match to succeed, except "–" indicates the ANDNOT operation.

An expression enclosed in a parentheses and starting with a literal [e.g., (COUNT) in production 4] will match any expression in STM which starts with the same literal [e.g., (COUNT 2)]. The system attempts to match the rules in the order shown and starts over from the beginning of the set after a successful match.

REP — REPlace, so that, e.g., the RHS of production 1 will replace the expression (READY) in the data base with the expression (COUNT X1) [where the variable X1 stands for the element matched by the X1 in (ORDER X1)]

DEP — DEPosit symbols at front of STM

ATTEND — wait for input from tty and interpret it as an ACTION to be carried out. For this example, typing (DEP(M 4) (N 2)) will deposit those symbols in STM and have the system add 4 and 2.

SAY — output to tty

(COND...) — shorthand for (DEP (COND...))

(ACTION...) — shorthand for (DEP (ACTION...))

PROD — gather all items in the STM of the form (COND...) and put them together into a LHS; gather all items of the form (ACTION...) and put them together into a RHS; removing all these expressions from the STM. Form a production from the resulting LHS and RHS, and add it to the front of the set of productions (i.e., before rule 1). (see note [11] for a comment on the importance of rule order)

Figure 2a

(after [Waterman, 1974], simplified slightly)

comments on system behavior

The "S" (rules 2, 4 and 5) is intended to indicate the successor function.

After initialization (rules 1 and 2), the system loops around rules 4 and 5, incrementing NN by 1 forM interations. In this loop, intermediate calculations (the results of successor function computations) are saved via the (PROD) in rule 5, and the final answer is saved by the (PROD) in rule 3. Thus, after computing 4 + 2, the rule set will contain the additional rules:

```
(S X3 0)      ⇒ (REP (S X3 0) (X3 1))
                 .
                 .
                 .
(S X3 5)      ⇒ (REP (S X3 5) (X3 6))
(M 4) (N 2)   ⇒ (SAY 6 IS THE ANSWER) (STOP)
```

The system is thus recording its intermediate and final results by writing new productions, and in the future will have these answers available a single step. Note that the set of productions therefore *is* memory (and in fact long term memory, LTM, since productions are never lost from the set).

Figure 2b

Constrained format

While there are wide variations in the format permitted by various PSs, in any given system the syntax is traditionally quite restrictive, and generally follows the conventions accepted for PSs[8]. Most commonly this means, first, that the side of the rule to be matched should be a simple predicate built out of a Boolean combination of computationally primitive operations, which involve (as noted above) only matching and detection. Second, it means the side of the rule to be executed should perform conceptually simple operations on the data base. In many of the systems oriented toward psychological modelling, the side to be matched consists of a set of literals or simple patterns, with the understanding that the set is to be taken as a conjunction, so that the predicate is an implicit one regarding the success or failure of matching all of the elements. Similarly, the side to be executed performs a simple symbol replacement or rearrangement.

Whatever the format, though, the conventions noted lead to clear restrictions for a pure production system. First, as a predicate, the matching side of a rule should return only some indication of the success or failure of the match (while binding individual variables or segments in the process of pattern matching is quite often used, it would be considered inappropriate to have the matching process produce a complex data structure intended for processing by another part of the system). Second, as a simple expression, the matching operation is precluded from using more complex control structures like iteration or recursion within the expression itself (such operations can be constructed from multiple rules, however). Finally, as a matching and detection operation, it must only 'observe' the state of the data base, and not change it in the operation of testing it.

We can characterize a continuum of possibilities for the side of the rule to be executed. There might be a single primitive action, a simple collection of independent actions, a carefully ordered sequence of actions, or yet more complex

control structures. We suggest that there are two related forms of simplicity which are important here. First, each action to be performed should be one which is a conceptual primitive for the domain. In the DENDRAL system, for example, it is appropriate to use chemical bond breaking as the primitive, rather than describing the process at some lower level. Second, the complexity of control flow for the execution of these primitives should be limited–in a 'pure' production system, for example, we might be wary of a complex set of actions that is, in effect, a small program of its own. Again, it should be noted that the system designer may of course follow or disregard these retrictions. The result, however, will conform to the traditional PS architecture to the extent that they are met.

These constraints on form make the dissection and 'understanding' of productions by other parts of the program a more straightforward task, strongly enhancing the possibility of having the program itself read, and/or modify (rewrite) its own productions[9]. Expressability suffers, however, since the limited syntax may not be sufficiently powerful to make expressing each piece of knowledge an easy task. This in turn both restricts extensibility (adding something is difficult if it's hard to express it), and makes modification of the system's behavior more difficult (e.g., it might not be particularly attractive to implement a desired iteration if it requires several rules rather than a line or two of code).

Rules as primitive actions

In a 'pure' PS, the smallest unit of behavior is a rule invocation, which, at its simplest, involves the matching of literals on the LHS, followed by replacement of those symbols in the data base with the ones found on the RHS. While the variations can be more complex, it is in some sense a violation of the spirit of things to have, e.g., a sequence of actions in the RHS.

Moran (Moran, 1973b), for example, acknowledges a deviation from the spirit of production systems in his VIS, when he groups rules in "procedures" within which the rules are totally ordered for the purpose of conflict resolution. He sees several advantages in this departure: it is "natural" for the user (a builder of psychological models) to write rules as a group working toward a single goal. This grouping restricts the context of the rules. It also helps minimize the problem of implicit context: when rules are ordered, a rule which occurs later in the list may really be applicable only if some of the conditions checked by earlier rules are untrue. This dependency, referred to as implicit context, is often not made explicit in the rule, but may be critical to system performance. The price paid for these advantages is twofold: first, extra rules, less directly attributable to psychological processes, are needed to switch among procedures; second, it violates the basic production system tenet that any rule should (in principle) be able to fire at any time–here only those in the currently active procedure can fire.

To the extent that the pure production system restrictions are met, we can consider rules as the quanta of intelligent behavior in the system. Otherwise, as

in the VIS sytem, we must look at larger aggregations of rules to trace behavior. In doing so we lose some of the ability to quantify and measure behavior, as is done, for example, with the PSG system simulation of the Sternberg task, where response times are attributed to individual production rules, and then compared against actual psychological data.

A different sort of deviation is found in the DENDRAL system, and in a few MYCIN rules. In both, the RHS is effectively a small program, carrying out complex sequences of actions. In this case, the quanta of behavior are the individual actions of these programs, and understanding the system thus requires familiarity with them.

By embodying these bits of behavior in a stylized format, it becomes possible for the system to 'read' them to its users (as is done in MYCIN, see [Davis, 1975] and note [9]), and hence provide some explanation of its behavior, at least at this level. This prohibition against complex behaviors within a rule, however, may force us to implement what are (conceptually) simple control structures by using the combined effects of several rules. This of course may make overall behavior of the system much more opaque (see "Visibility," below).

Modularity

We can regard the *modularity* of a program as the degree of separation of its functional units into isolatable pieces. A program is *highly modular* if any functional unit can be changed (added, deleted, or replaced) with no unanticipated change to other functional units. Thus program modularity is inversely related to the strength of coupling between its functional units.

The modularity of programs written as pure production systems arises from the important fact that the next rule to be invoked is determined solely by the contents of the data base, and no rule is ever called directly. Thus the addition (or deletion) of a rule does not require the modification of any other rule to provide for (delete) a call to it. We might demonstrate this by repeatedly removing rules from a PS: many systems will continue to display some sort of "reasonable" behavior, up to a point[10]. By contrast, adding a procedure to an ALGOL-like program requires modification of other parts of the code to insure that it is invoked, while removing an arbitary procedure from such a program will generally cripple it.

Note that the issue here is more than simply the 'undefined function' error message which would result from a missing procedure. The problem persists even if the compiler or interpreter were altered to treat undefined functions as no-ops. The issue is a much more fundamental one concerning organization of knowledge: programs written in procedure-oriented languages stress the kind of explicit passing of control from one section of code to another that is characterized by the calling of procedures. This is typically done at a selected time and in a particular context, both carefully chosen by the programmer. If a no-op is substituted for a missing procedure, the context upon returning will not be what

the programmer expected, and subsequent procedure calls will be executed in increasingly incorrect environments. Similarly, procedures which have been added must be called from *somewhere* in the program, but the location of the call must be chosen carefully if the effect is to be meaningful.

Production systems, on the other hand, especially in their pure form, emphasize the decoupling of control flow from the writing of rules. Each rule is designed to be ideally, an independent chunk of knowledge with its own statements of relevance (either the conditions of the LHS, as in a data-driven system, or the action of the RHS, as in a goal-directed system). Thus where the ALGOL programmer carefully chooses the order of procedure calls to create a selected sequence of environments, in a production system it is the environment which chooses the next rule for execution. And since a rule can only be chosen if its criteria of relevance have been met, the choice will continue to be a plausible one, and system behavior remain "reasonable," even as rules are successively deleted.

This inherent modularity of pure production systems eases the task of programming in them. Given some primitive action that the system fails to perform, it becomes a matter of writing a rule whose LHS matches the relevant indicators in the data base, and whose RHS performs the action. Where the task is then complete for a pure PS, systems which vary from this design have the additional task of assuring proper invocation of the rule (not unlike assuring the proper call of a new procedure). The difficulty of this varies from trivial in the case of systems with goal oriented behavior (like MYCIN), to substantial in systems that use more complex LHS scans and conflict resolution strategies.

For systems using the goal-oriented approach, rule order is usually unimportant. Insertion of a new rule is thus simple, and can often be totally automated. This is of course a distinct advantage where the rule set is large, and the problems of system complexity are significant.

For others (like PSG and PASII) rule order can be critical to performance and hence requires careful attention. This can, however, be viewed as an advantage, and indeed, Newell (Newell, 1973) tests different theories of behavior by the simple expedient of changing the order of rules. The family of Sternberg task simulators there includes a number of production systems which differ only by the interchange of two rules, yet display very different behavior. Waterman's system (Waterman, 1974) accomplishes 'adaptation' by the simple heuristic of placing a new rule immediately before a rule that causes an error[11].

Visibility of behavior flow

Visibility of behavior flow is the ease with which the overall behavior of a PS can be understood, either by observing the system, or by reviewing its rule base. Even for conceptually simple tasks, the stepwise behavior of a PS is often rather opaque. The poor visibility of PS behavior compared to that of the procedural formalism is illustrated by the Waterman arithmetic example. The procedural version of the iterative loop there is reasonably clear (lines B, C and E), and an

ALGOL-type FOR I:=1 UNTIL N DO... would be completely obvious. Yet the PS formalism for the same thing requires non-intuitive productions (like 1 and 2), and symbols like NN whose only purpose is to "mask" the condition portion of a rule so it will not be invoked later (such symbols are termed *control elements* [Anderson, 1976]).

The requirement for control elements and much of the opacity of PS behavior is a direct result of two factors noted above: the unity of control and data store, and the reevaluation of the data base at every cycle. Because of these, any attempt to 'read' a PS requires keeping in mind the entire contents of the data base, and scanning the entire rule set at every cycle. Control is much more explicit and localized in procedural languages, so that reading ALGOL code is a far easier task (Indeed, one of the motivations for the current interest in structured programming is the attempt to emphasize still further the degree of explicitness and localization of control).

The perspective on knowledge representation suggested by PSs also contributes to this opacity. As suggested above, PSs are appropriate when it is possible to specify the content of required knowledge, without also specifying the way in which it is to be used. Thus, reading a PS does not generally make clear how it works, so much as what it may know, and the behavior is consequently obscured. The situation is often reversed in procedural languages—program behavior may be reasonably clear, but the domain knowledge used is often opaquely embedded in the procedures. The two methodologies thus emphasize different aspects of knowledge and program organization.

Machine readable

Several interesting capabilities arise from making it possible for the system to examine its own rules. As one example, it becomes possible to implement automatic consistency checking. This can proceed at several levels: in the simplest approach we can search for straightforward 'syntactic' problems such as contradictions (e.g., 2 rules of the form $A \wedge B \rightarrow C$ and $A \wedge B \rightarrow -C$), or subsumption (e.g., $D \wedge E \wedge F \rightarrow G$, $D \wedge F \rightarrow G$). A more sophisticated approach, which would require extensive domain-specific knowledge, might be able to detect 'semantic' problems, as for example a rule of the form $A \wedge B \rightarrow C$, when it is known from the meaning of A and B that $A \rightarrow -B$. Many other (domain specific) tests may also be possible. The point is that by automating the process, extensive (perhaps exhaustive) checks of newly added productions are possible (and could perhaps be run in background mode when the system is otherwise idle).

A second sort of capability is described in note [9], and deals with the MYCIN system's approach to examining its rules. This is used in several ways (Davis, 1976), and produces both a more efficient control structure, and precise explanations of system behavior.

Explanation of primitive actions

Production system rules are intended to be modular chunks of knowledge and

to represent primitive actions. Thus, explaining primitive acts should be as simple as stating the corresponding rule—all necessary contextual information should be included in the rule itself. Achieving such clear explanations, however, evidently strongly depends upon the extent to which the assumptions of modularity and explicit context are met. In the case where stating a rule does provide a clear explanation, the task of modification of program behavior becomes easier.

As an example, the MYCIN system often successfully uses rules to explain its behavior (see [Davis, 1975] and [Shortliffe, 1975a] for examples). This form of explanation fails, however, when considerations of system performance or human engineering lead to rules whose context is obscure. One rule, for example, says in effect, "If A seems to be true, and B seems to be true, then that's (more) evidence in favor of A." It is phrased this way rather than simply "if B seems true, that's evidence in favor of A," because B is a very rare condition, and it appears counterintuitive to ask about it unless you suspect A to begin with. The first clause of the rule is thus acting as a strategic filter, to insure that the rule is not even tried unless it has a reasonable chance of succeeding. System performance has been improved (especially as regards human engineering considerations), at the cost of a somewhat more opaque rule.

Modifiability, consistency, rule selection mechanism

As noted above, the tightly constrained format of rules makes it possible for the system to examine its own rule base, with the possibility of modifying it in response to requests from the user, or to insure consistency with respect to newly added rules. While all these are conceivable in a system using a standard procedural approach, it is the heavily stylized format of rules and the typically simple control structure of the interpreters that makes them all realizable prospects in a PS.

Finally, the relative complexity of the rule selection mechanism will have varying effects on the ability to automate consistency checks, or behavior modification and extension. A RHS scan with backward chaining seems to be the easiest to follow since it mimics part of human reasoning behavior, while a LHS scan with a complex conflict resolution strategy makes the system generally more difficult to understand. As a result, predicting and controlling the effects of changes in, or additions to, the rule base are directly influenced in either direction by the choice of rule selection method.

Programmability

It is hard to imagine any factor in this section which does not interact with programmability (and it has therefore been omitted from Figure 1). In our experience, the answer to "how easy is it to program in this formalism?" is "it's reasonably difficult." This experience is apparently not unique:

> Any structure which is added to the system diminishes the explicitness of rule conditions.... Thus rules acquire implicit conditions.

> This makes them (superficially) more concise, but at the price of clarity and precision.... Another questionable device in most present production systems (including mine) is the use of tags, markers, and other cute conventions for communicating between rules. Again, this makes for conciseness, but it obscures the meaning of what is intended. The consequence of this in my program is that it is very delicate: one little slip with a tag and it goes off the track. Also, it is very difficult to alter the program; it takes a lot of time to readjust the signals.
>
> (Moran, 1973a)

One source of the difficulties in programming production systems is the necessity, referred to above, of programming "by side effect." Another is the difficulty of using the PS methodology on a problem that cannot be broken down into the solution of independent subproblems, or into the synthesis of a behavior which is neatly decomposable.

Several techniques have been investigated to deal with this difficulty. One of them is the use of 'tags and markers' (control elements) referred to above. These can be used in various ways, and we have come to believe that the manner in which they are used, particularly in the psychological modelling systems, can be an indication of how successfully the problem has been put into PS terms.

To demonstrate this, consider two very different (and somewhat idealized) approaches to writing a PS. In the first, the programmer writes each rule independently of all the others, simply attempting to capture in each some chunk of required knowledge. The creation of each rule is thus a separate task. Only when all of them have been written are they assembled, the data base is initialized, and the behavior produced by the entire set of rules is noted.

As a second approach, the programmer starts out with a specific behavior which he wants to recreate. The entire rule set is written as a group with this in mind, and, where necessary, one rule might deposit a symbol like A00124 in STM solely to trigger a second specific rule on the next cycle.

In the first case, the control elements would correspond to recognizable states of the system. As such, they function as indicators of those states and serve to trigger what is generally a large class of potentially applicable rules[12]. In the second case there is no such correspondence, and often only a single rule recognizes a given control element. The idea here is to insure the execution of a specific sequence of rules, often because a desired effect could not be accomplished in a single rule invocation.

Such idiosyncratic use of control elements is formally equivalent to allowing one rule to call a second, specific rule, and hence is very much out of character for a PS. To the extent that it is used, it appears to us to be suggestive of a failure of the methodology—perhaps because a PS was ill-suited to the task to begin with, or because the particular decomposition used for the task was not well chosen. (The possibility remains, of course, that a 'natural' interpretation of

a control element will be forthcoming as the model develops, and the additional rules which refer to it will be added. In that case the ease of adding the new rules arises out of the fact that the technique of allowing one rule to call another explicitly was not used.) Since one fundamental assumption of the PS methodology as a psychological modelling tool is that states of the system correspond to what are at least plausible (if not immediately recognizable) individual "states of mind," the relative abundance of the two uses of control elements mentioned above can conceivably be taken as an indication of how successfully the methodology has been applied.

A second approach to dealing with the difficulty of programming in PSs is the use of increasingly complex forms within a single rule. Where a 'pure' PS might have a single action in its RHS, many of the current psychological modelling systems (PAS II, VIS) have explored the use of more complex sequences of actions, including the use of conditional exits from the sequence.

Finally, one recent effort (Rychener, 1975) has investigated the use of PSs which are unconstrained by prior restrictions on rule format, use of tags, etc. The aim here is to employ the methodology as a formalism for explicating knowledge sources, understanding control structures, and examining the effectiveness of PSs for attacking the large problems typical of artificial intelligence. The productions in this system often turn out to have a relatively simple format, but complex control structures are built via carefully orchestrated interaction of rules. This is done with several techniques, including explicit reliance on both control elements and certain characteristics of the data base architecture. For example, iterative loops are manufactured via explicit use of control elements, and data is (redundantly) re-asserted in order to make use of the 'recency' ordering on rules (the rule which mentions the most recently asserted data item is chosen first; see "Architecture," below). These techniques have supported the reincarnation into PSs of a number of sizable AI programs (e.g., Bobrow's STUDENT [Bobrow, 1968]), but, as the author notes, "control tends to be rather inflexible, failing to take advantage of the openness that seems to be inherent in PSs."

This reflects something of a new perspective on the use of PSs. Previous efforts have used them as tools for analyzing both the core of knowledge essential to a given task, and the manner in which such knowledge is used. Such efforts relied in part on the austerity of the available control structure to keep all the knowledge explicit (recall Moran's comment above). The expectation is that each production will embody a single chunk of knowledge. Even in the work of (Newell, 1973), which used PSs as a medium for expressing different theories (= different control structures) in the Sternberg task, an important emphasis is placed on productions as a model of the detailed control structure of humans. In fact, "every aspect of the system" is assumed to have a psychological correlate (Newell, 1973, p. 472).

The work reported in (Rychener, 1975), however, after explicitly detailing the chunks of knowledge required in the word problem domain of STUDENT,

notes a many-to-many mapping between its knowledge chunks and productions. It also focusses on complex control regimes which can be built using PSs. While still concerned with knowledge extraction and explication, it views PSs more as an abstract programming language and uses them as a vehicle for exploring control structures. While this approach does offer an interesting perspective on such issues, it should also be noted that as productions and their interactions grow more complex, many of the advantages associated with 'traditional' PS architecture may be lost (as for example, the loss of openness noted above). The benefits to be gained are roughly analogous to those of using a higher level programming language: while the finer grain of the process being examined may become less obvious, the power of the language permits far larger scale tasks to be undertaken, and makes it easier to examine larger scale phenomena like the interaction of entire categories of knowledge.

The use of PS has thus grown to encompass many different forms, many of which are far more complex than the 'pure' PS model described initially.

TAXONOMY FOR PRODUCTION SYSTEMS

In this section we suggest four dimensions along which to characterize PSs, examine related issues for each, and indicate the range of each dimension as evidenced by systems currently (or recently) in operation.

Form — how primitive or complex should the syntax of each side be?

There is a wide variation in syntax used by various systems, and corresponding differences in both the matching & detection process, and the subsequent action caused by rule invocation. For matching, in the simplest case only literals are allowed, and it is a conceptually trivial process (although the rule and data base may be so large that efficiency becomes a consideration). Successively more complex approaches allow free variables (Waterman's poker player [Waterman, 1970]), syntactic classes (as in some parsing systems), and increasingly sophisticated capabilities of variable and segment binding, and pattern specification (PAS II, VIS, LISP70; for an especially thorough discussion of pattern matching methods in production systems as used in VIS, see [Moran, 1973a], pp. 42-45).

The content of the data base also influences the question of form. One interesting example is Anderson's ACT system (Anderson, 1976), whose rules have node networks in their LHS. The appearance of an additional piece of network as input results in a "spread of activation" occurring in parallel through the LHS of each production. The rule that is chosen is the one whose LHS most closely matches the input and which has the largest subpiece of network already in its working memory.

As another example, the DENDRAL system uses a literal pattern match, but its patterns are graphs representing chemical classes, and can be quite complex. Each class is defined by a basic chemical structure, referred to as a *skeleton*. As in the data base, atoms composing the skeleton are given unique numbers, and

chemical bonds are described by the numbers of the atoms they join [e.g., (5 6)]. The LHS of a rule is the name of one of these skeletons, and a side effect of a successful match is the recording of the correspondence between atoms in the skeleton and those in the molecule.

The action parts of these rules describe a sequence of actions to perform: *break* one or more bonds, saving a molecular fragment, and *transfer* one or more hydrogen atoms from one fragment to another. An example of a simple rule is:

```
ESTROGEN ⇒  (BREAK (14 15) (13 17))
            (HTRANS +1 +2)
```

The LHS here is the name of the graph structure which describes the estrogen class of molecules, while the RHS indicates the likely locations for bond breakages and hydrogen transfers when such molecules are subjected to mass spectral bombardment. Note that while both sides of the rule are relatively complex, they are written in terms which are conceptual primitives in the domain.

A related issue is illustrated by the rules used by MYCIN, where the LHS consists of a Boolean combination of standardized predicate functions. Here the testing of a rule for relevance consists of having the standard LISP evaluator evaluate the LHS, and all matching and detection is controlled by the functions themselves. While there is power available in using functions that is missing from a simple pattern match, there is also the temptation of writing one function to do what should have been expressed by several rules.

For example, one small task in MYCIN is to deduce that certain organisms are present, even though they have not been recovered from any culture. This is a conceptually complex, multi-step operation, which is currently handled by invocation of a single function. (Work is underway in MYCIN to provide a much cleaner, rule based solution, which will allow easier access and modification of the knowledge required for the task).

If one succumbs often to the temptation to write one function rather than several rules, the result can be a system that may perform the initial task, but which loses a great many of the other advantages of the PS approach. The problem is that the knowledge embodied in these functions is unavailable to anything else in the system. Where rules can be accessed and their knowledge examined (because of their constrained format), chunks of ALGOL-like code are not nearly as informative. The availability of a standardized, well structured set of operational primitives can help to avoid the temptation to create new functions unnecessarily.

Content — How 'far' conceptually is it from the LHS to the RHS? Which conceptual levels of knowledge belong in rules?

The question here is how large a reasoning step is to be embodied in a single rule, and there seem to be two distinct approaches. Systems designed for psychological modelling (PAS II, PSG, etc.), try to measure and compare tasks and

determine required knowledge and skills. As a result, they try to dissect cognition into its most primitive terms. While there is, of course, a range of possibilities, from the simple literal replacement of PSG to the more sophisticated abilities of PAS II to construct new productions, rules in these systems tend to embody only the most basic conceptual steps.

Grouped at the other end of this spectrum are the 'task oriented' systems like DENDRAL and MYCIN, which are designed to be competent at selected real world problems. Here the conceptual primitives are at a much higher level, encompassing in a single rule a piece of reasoning which may be based both on experience and a highly complex model of the domain. For example, the statement that "a gram negative rod in the blood is likely to be an E. coli" is based in part on a knowledge of physiological systems, and in part on clinical experience. Often the reasoning step is sufficiently large that the rule becomes a significant statement of a fact or principle in the domain, and, especially where reasoning is not yet highly formalized, a comprehensive collection of such rules may represent a substantial portion of the knowledge in the field.

An interesting, related point of methodology is the question of what kinds of knowledge ought to go into rules. Rules expressing knowledge about the domain are the necessary initial step, but interest has been generated lately in the question of embodying strategies in rules. One of us has been actively pursuing this in the implementation of *meta-rules* in the MYCIN system (Davis, 1976). These are rules about rules, and contain strategies and heuristics. Thus while the ordinary rules contain standard object-level knowledge about the medical domain, meta-rules contain information about rules, and embody strategies for selecting potentially useful paths of reasoning. For example, a meta-rule might suggest that

> if the patient has had an ulcer, then in concluding about organism identity, rules which mention the gastro-intestinal trace are more likely to be useful

There is clearly no reason to stop at one level, however—third order rules could be used to select from or order the meta-rules, by using information about how to select a strategy (and hence represent a search through "strategy space"); fourth order rules would suggest how to select criteria for choosing a strategy, etc.

This approach appears to be promising for several reasons. First, the expression of any new level of knowledge in the system can mean in increase in competence. This sort of strategy information, moreover, may translate rather directly into increased speed (since fewer rules need be tried), or equivalently, no degradation in speed even with large increases in the number of rules. Second, since meta-rules refer to rule content rather than rule names, they automatically take care of new object level rules that may be added to the system. Third, the possibility of expressing this information in a format that is essentially the same as the standard one means a uniform expression of many levels of knowledge.

This uniformity in turn means that the advantages which arise out of the embodiment of any knowledge in a production rule (accessibility, and the possibility of automated explanations, modifications, and acquisition of rules) should be available for the higher order rules as well.

Control cycle architecture[13]

The basic control cycle can be broken down into two phases called *recognition* and *action*. The recognition phase involves selecting a single rule for execution, and can be further subdivided into *selection* and *conflict resolution*. In the selection process, one or more potentially applicable rules are chosen from the set, and passed to the conflict resolution algorithm, which chooses one of them.

There are several approaches to *selection*, which can be categorized by their rule scan method. Most systems (e.g., PSG, PAS II) use some variation of a LHS scan, in which each LHS is evaluated in turn. Many stop scanning at the first successful evaluation (e.g., PSG), and hence conflict resolution becomes a trivial step (although the question then remains of where to start the scan on the next cycle: start over at the first rule, or continue from the current rule).

Some systems, however, collect all rules whose LHS's evaluate successfully. Conflict resolution then requires some criterion for choosing a single rule from this set (called the conflict set). Several have been suggested, and include:

i) rule order–there is a complete ordering of all rules in the system, and the rule in the conflict set with the highest priority is chosen
ii) data order–elements of the data base are ordered, and that rule is chosen which matches element(s) in the data base with highest priority
iii) generality order–the most specific rule is chosen
iv) rule precedence–a precedence network (perhaps containing cycles) determines the hierarchy
v) recency order–choosing either the most recently executed rule, or the rule containing the most recently matched element of the data base.

For example, the LISP70 interpreter uses (iii), while DENDRAL uses (iv).

A different approach to the selection process is used in the MYCIN system. It is goal-oriented, and uses a RHS scan. The process is quite similar to the unwinding of consequent theorems in PLANNER (Hewitt, 1972)–given a required subgoal, the system retrieves the (unordered) set of rules whose actions conclude something about that subgoal. The evaluation of the first LHS is begun, and if any clause in it refers to a fact not yet in the data base, a generalized version of this fact becomes the new subgoal, and the process recurs. However, because MYCIN is designed to work with judgmental knowledge in a domain (clinical medicine) where collecting all relevant data and considering all possibilities are very important, it in general executes *all* rules from the conflict set, rather than stopping after the first success.

The meta-rules mentioned above may also be seen as a way of selecting a subset of the conflict set for execution. There are several interesting advantages to this. First, the conflict resolution algorithm is stated explicitly in the meta-rules (rather than implicitly in the system's interpreter), and in the same representation as the rest of the rule-based knowledge.

Second, since there can be a set of meta-rules for each subgoal type, MYCIN can specify distinct, and hence potentially more customized conflict resolution strategies for each individual subgoal. Since the backward chaining of rules may also be viewed as a depth first search of an AND/OR goal tree, we have the appearance of doing a tree search through a tree with an interesting property—a collection of specific heuristics about which path to take may be stored at every branch point in the tree.

In addition, rules in the system are inexact, judgmental rules with a model of "approximate implication" in which the user may specify a measure of how firmly he believes that a given LHS implies its RHS (Shortliffe, 1975b). This admits the possibility of writing numerous, perhaps conflicting heuristics, whose *combined* judgment forms the conflict resolution algorithm.

Control cycle architecture affects the rest of the production system in various ways. Overall efficiency, for example, can be strongly influenced. The RHS scan in a goal-oriented system insures that only relevant rules are considered in the conflict set. Since this is often a small subset of the total, and one which can be computed once and stored for reference, there is no search necessary at execution time, so the approach can be quite efficient. (In addition, since this approach seems natural to humans, the system's behavior becomes easier to follow).

Among the conflict resolution algorithms mentioned, rule order and recency order require a minimal amount of checking to determine the rule with highest priority. The generality order can be efficiently implemented, and in fact the LISP70 compiler uses it quite effectively. Data order and rule precedence require a significant amount of bookkeeping and processing, and hence may be slower (PSH, a recent development along the lines of PSG, attacks precisely this problem.)

The relative difficulty of adding a new rule to the system is also determined to a significant degree by the choice of control cycle architecture. Like PLANNER with its consequent theorems, the goal oriented approach makes it possible to simply "throw the rule in the pot" and still be assured that it will be retrieved properly. The generality ordering technique also admits of a simple automatic method for placing the new rule, as do the data ordering and recency strategy. In the latter two cases, however, the primary factor in ordering is external to the rule, and hence while they may be added to the rule set easily, it is somewhat harder to predict and control their subsequent selection. For both complete rule order and rule precedence networks, rule addition may be a substantially more difficult problem, and depends primarily on the complexity of the criteria used to determine the hierarchy.

System augmentability, extensibility

Learning, viewed as augmentation of the system's rule base, is of concern to both the information processing psychologists, who view it as an essential aspect of human cognition, and designers of knowledge-based systems, who acknowledge that building truly expert systems requires an incremental approach to competence. As yet we have no range or even points of comparison to offer, because of the scarcity of examples. Instead we suggest some standards by which the ease of augmentation may be judged. (It should be noted, however, that this discussion is oriented primarily toward an interactive mixed initiative view of learning, in which the human expert teaches the system, and answers questions it may generate. It has also been influenced by the experience of one of us in attacking the problem for the MYCIN system [Davis, 1976]. Many other models of the process [e.g., teaching by selected examples] are of course possible, and would most likely require different sets of criteria).

Perhaps the most basic question is "How automatic is it?" The ability to learn is clearly an area of competence by itself, and thus we are really asking how much of that competence has been captured in the system, and how much the user has to supply. Some aspects of this competence include:

> if the current system displays evidence of a bug caused by a missing or incorrect rule, how much of the diagnosing of the bug is handled by the system, and how much tracing must be done by the user?
>
> once the bug is uncovered, who fixes it? Must the user modify the code by hand; tell the system in some command language what to do; indicate the generic type of the error; can he simply point out the offending rule, or can the system locate and fix the bug itself?
>
> can the system indicate if the new rule will in fact fix the bug, or whether it will have side effects or undesired interactions?
>
> how much must the user know about rule format conventions when expressing a new (or modified) rule? Must he know how to code it explicitly; know precisely the vocabulary to use; know generally how to phrase it; or can he indicate in some general way the desired rule and allow the system to make the transformation? Who has to know the semantics of the domain? For example, can the system detect impossible conjunctions [$A \wedge B$, where $A \equiv$ not-B, but 'equivalent' in the semantic sense], or trivial disjunctions [$A \vee B$, where $A \equiv$ not-B]? Who knows enough about the system's idiosyncracies to suggest optimally fast or powerful ways of expressing rules?
>
> how hard is it to enter strategies?
>
> how hard is it to enter control structure information? Where is the control structure information stored: in aggregations of rules, or

in higher order rules? The former makes augmentation or modification a difficult problem, the latter makes it somewhat easier, since the information is explicit, and concentrated in one place.

can you assure continued consistency of the rule base? Who has to do the checking?

We believe these are questions that will be important and useful to confront in designing almost any system intended to do knowledge acquisition, and especially for those built around production rules as knowledge representation.

CONCLUSIONS

In artificial intelligence research, production systems were first used as a means of embodying primitive chunks of information processing behavior in simulation programs. Their adaptation to other uses, and increased experience with them has focussed attention on their possible utility as a general programming mechanism. Production systems permit the representation of knowledge in a highly uniform and modular way. This may pay off handsomely in two areas of investigation–development of programs that can manipulate their own representations, and development of a theory of loosely coupled systems, both computational and psychological. Production systems are potentially useful as a flexible modelling tool for many types of systems; current research efforts are sufficiently diverse to discover the extent to which this potential may be realized.

Information processing psychologists continue to be interested in production systems for many reasons. PSs can be used to study a wide range of tasks; they constitute a general programming system with the full power of a Turing Machine, but use a homogeneous encoding of knowledge; to the extent that the methodology is that of a pure production system, the knowledge embedded is completely explicit, and thus aids experimental verification or falsification of theories which use PSs as a medium of expression; productions may correspond to verifiable bits of psychological behavior (Moran, 1973a), reflecting the role of postulated human information processing structures such as short term memory; they are flexible enough to permit a wide range of variation based on reaction times, adaptation, or other commonly tested psychological variables; the space of PS can be fit to various alternative human information processing strategies; and finally, they provide a method for studying learning and adaptive behavior (Waterman, 1974).

For those wishing to build knowledge-based expert systems, the homogeneous encoding of knowledge offers the possibility of automating parts of the task of dealing with the growing complexity of such systems. Knowledge in production rules is both accessible and relatively easy to modify. It can be executed by one part of the system as procedural code, and examined by another part as if it were a declarative expression. Despite the difficulties of programming PSs, and their occasionally restrictive syntax, the fundamental

methodology at times suggests a convenient and appropriate framework for the task of structuring and specifying large amounts of knowledge. It may thus prove to be of great utility in dealing with the problems of complexity encountered in the construction of large knowledge bases.

ACKNOWLEDGMENTS

The creation and development of this paper has extended for several months, and has profited greatly from advice from many quarters. We are indebted to Bruce Buchanan for several careful readings and quite a few extended discussions. Many helpful suggestions were offered by Allen Newell, Ed Feigenbaum, Don Waterman, Tom Moran, Terry Winograd, and Stu Card, who were patient enough to help debug early drafts. The work reported here was funded in part by grants from the Health Resources Administration, Grant HS01544, and the Advanced Research Projects Agency ARPA Contract DAHC15-73-C-0435.

NOTES

1 Some of the observations reported here are insights gained by listening to the speakers in a seminar held at Stanford in late 1974, and we have tried to acknowledge those sources as accurately as possible. Apologies are offered in advance for any omissions; unattributed opinions are those of the authors.

Despite our best efforts, inevitable biases appear, so we may as well be explicit: our experience with production systems as high performance application programs is more extensive than that with the psychological modelling applications, and this is no doubt apparent at times. We have done our best to minimize such occurences.

2 Newell, A., Simon, H., *Human Problem Solving*, p. 44, Prentice-Hall, 1972.

3 One class of production systems we will not attempt to treat at any length here is their use as grammars for formal languages. While the intellectual roots are similar (Floyd, 1961; Evans, 1964), their use has evolved a distinctly different flavor. In particular, their use of non-determinism is an important factor which provides a very different perspective on issues of control structure and effectively renders the question of rule selection a moot point.

4 For example:

> The critical evaluation of EPAM must ultimately depend not upon the interest which it may have as a learning machine, but upon its ability to explain and predict phenomena of verbal learning.
>
> (Feigenbaum, 1963)

These phenomena included stimulus and response generalization, oscillation, retroactive inhibition, and forgetting, all of which are 'mistakes' for a system intended for high performance, but are important in a system meant to model human learning behavior.

5 Newell, A., Remarks on the relationship betwen artificial intelligence and cognitive psychology, in *Theoretical Approaches to Non-Numerical Problem Solving*, Part IV, pp. 363-400, Springer-Verlag, 1970.

6 E. Feigenbaum, private communication.

7 As noted in (Waterman, 1974), the production rule version does not assume the existence of a successor function (as the procedural version does); instead rule 5 writes new productions that give the successor for specific integers. Rule 3 builds what amounts to an addition table, writing a new production for each example the system is given. Placing these new rules at the 'front' of the rule set (i.e., before rule 1) means that the addition table and successor function table will always be consulted before a computation is attempted, and the answers obtained in one step if

possible. Without these extra steps, and with a successor function, the production rule set could be smaller and hence slightly less complex.

8 Note, however, that the tradition arises out of a commonly followed convention rather than any essential characteristic of a PS.

9 The current MYCIN system makes strong use of the concept of allowing one part to the system to 'read' the rules being executed by another part. As one example, the system does a partial evaluation of rule premises. Since a premise is a Boolean combination of predicate functions like

```
($AND(SAME CNTXT SITE BLOOD)      {the site of the culture is blood and
   (SAME CNTXT GRAM GRAMPOS)       the gramstain is grampositive and
   (DEFIS CNTXT AIR AEROBIC))      the aerobicity is definitely aerobic}
```

and since clauses which are unknown cause subproblems to be set up which may involve long computations, it makes sense to check to see if, based on what is currently known, the entire premise is sure to fail (e.g., if any clause of a conjunction is known to be false). We cannot simply EVAL each clause, since this will trigger a search if the value is still unknown. But if the clause can be 'unpacked' into its proper constituents, it is possible to determine whether or not the value is known as yet, and if so, what it is. This is done via a TEMPLATE associated with each predicate function. For example, the TEMPLATE for SAME is

(SAME CNTXT PARM VALU)

and it gives the generic type and order of arguments to the function (much like a simplified procedure declaration). By using this as a guide to unpack and extract the needed items, we can safely do a partial evaluation of the rule premise. A similar technique is used to separate the known and unknown clauses of a rule for the user's benefit when the system is explaining itself (see [Davis, 1976] for several examples).

Note that, first, part of the system is 'reading' the code being executed by the other part, and second, that this reading is guided by information carried in the rule components themselves. This latter characteristic assures that the capability is unaffected by the addition of new rules or predicate functions to the system.

10 How many rules could be removed without performance degradation (short of redundancies) is an interesting characteristic, which would appear to be correlated with the issue of which of the two common approaches to PSs is taken. The psychological modelling systems would apparently degenerate fastest, since they are designed to be minimally competent sets of rules. Knowledge-based expert systems, on the other hand, tend to embody numerous independent subproblems in rules, and often contain overlapping or even purposefully redundant representations of knowledge. Hence while losing their competence on selected problems, it appears they would still function reasonably well even with several rules removed.

11 One specific example of the importance of rule order can be seen in our earlier example of addition (Figure 2a). Here Rule 5 assumes that an ordering of the digits exists in STM in the form

(ORDER 0 1 2 . . .)

and from this can be created the successor function for each digit. If Rule 5 were placed before Rule 1, the system wouldn't add at all. In addition, acquiring the notion of successor in subsequent runs depends entirely on the placement of the new successor productions *before* Rule 3, or the effect of this new knowledge would be masked.

12 This basic technique of 'broadcasting' of information and allowing individual segments of the system to determine their own relevance has been extended and generalized in systems like HEARSAY II (Lesser, 1974), and the BEINGS system of Lenat (Lenat, 1975).

13 The range of conflict resolution algorithms in this section was suggested in a talk at the seminar by Don Waterman.

REFERENCES

Anderson, J., (1976) *Language, Memory and Thought,* L. Erlbaum Associates, (in preparation).

Bobrow, D.G. (1968) Natural language input for a computer problem-solving system, in *Semantic Information Processing,* (Minsky ed.), pp. 146-226.

Davis, R., Buchanan, B.G., Shortliffe, E.H. (1975) Production rules as a representation for a knowledge-based consultation program, *Artificial Intelligence,* (to appear, Jan. 77), Also, *AI Memo 266,* Computer Science Department, Stanford University.

Davis, R. (1976) Applications of meta level knowledge to the construction, maintenance, and use of large knowledge bases, Ph.D. thesis, *AI Memo 283,* Computer Science Department, Stanford University.

Evans, A. (1964) An ALGOL 60 compiler, in *Annual Review of Automatic Programming,* (Goodman, ed.), 4:87-124.

Feigenbaum, E.A. (1963) Simulation of verbal learning behavior, in *Computers and Thought,* (Feigenbaum and Feldman, eds.), McGraw Hill, pp. 297-309.

Feigenbaum, E.A., *et al.* (1971) On generality and problem solving–a case study involving the DENDRAL program, in *Machine Intelligence* 6, (Meltzer and Michie, eds.), pp. 165-190, Edinburgh University Press.

Floyd, R. (1961) A descriptive language for symbol manipulation, *JACM* 8:579-584.

Hewitt, C. (1972) Description and theoretical analysis of PLANNER, Ph.D. thesis, Department of Mathematics, MIT, Cambridge, Mass.

Lenat, D.L. (1975) BEINGS: Knowledge as interacting experts, *Proc. IJCAI 4* Tbilisi, USSR.

Lesser, R.L., *et al.* (1974) Organization of the HEARSAY II speech understanding system, *IEEE Symposium on Speech Recognition,* pp. 11-21.

Moran, T.P. (1973a) *The Symbolic Imagery Hypothesis: A Production System Model,* Computer Science Department, Carnegie-Mellon University.

Moran, T.P. (1973b) The symbolic nature of visual imagery, *Proc IJCAI 3,* Stanford, California.

Newell, A., Simon, H. (1972) *Human Problem Solving,* Prentice-Hall.

Newell, A. (1973) Production systems: Models of control structures, in *Visual Information Processing,* (William G. Chase, ed.), pp. 463-526, Academic Press.

Post, E. (1943) Formal reductions of the general combinatorial problem, *Am Jnl Math* 65:197-268. [For an interesting introduction to the general principles involved, see Minsky, *Computation: Finite and Infinite Machines,* Prentice-Hall, 1967, Chapter 12.]

Rychener, M.D. (1975) The studnt [sic] production system: a study of encoding knowledge in production systems, Technical Report (in press), Computer Science Department, Carnegie-Mellon University.

Shortliffe, E., *et al.* (1975a) Computer-based consultations in clinical therapeutics–explanation and rule acquisition capabilities of the MYCIN system, *Computers and Biomedical Research* 8, pp. 303-320.

Shortliffe, E., Buchanan, B. (1975b) A model of inexact reasoning in medicine, *Mathematical Biosciences* 23, pp. 351-379.

Smith, D.H., *et al.* (1972) Applications of artificial intelligence for chemical inference VIII *Journal of the American Chemical Society.* 94, p. 5962 ff.

Tesler, L.G., Enea, H.J., Smith, D.C. (1973) The LISP70 pattern matching system, *Proc IJCAI 3,* Stanford, California, pp. 671-676.

Waterman, D.A. (1970) Generalization learning techniques for automating the learning of heuristics *Artificial Intelligence* 1:121-170.

Waterman, D.A. (1974) *Adaptive Production Systems*, Complex Information Processing Working Paper No. 285, Psychology Department, Carnegie-Mellon University.

Winograd, T. (1975) Frame representations and the procedural/declarative controversy, in *Representation and Understanding*, (Bobrow and Collins, eds.), Academic Press.

DIALOGUE-TRANSFER OF KNOWLEDGE TO MACHINES

17

A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program-Understanding System

Cordell Green and David Barstow

Department of Computer Science
Stanford University

A hypothetical dialogue with a fictitious program-understanding system is presented. In the interactive dialogue the computer carries out a detailed synthesis of a simple insertion sort program for linked lists. The content, length and complexity of the dialogue reflect the underlying programming knowledge which would be required for a system to accomplish this task. The nature of the knowledge is discussed and the codification of such programming knowledge is suggested as a major research area in the development of program-understanding systems.

CONTENTS

INTRODUCTION

(a) SUMMARY
(b) DOMAIN OF DISCOURSE

A DIALOGUE

(a) INTRODUCTION
(b) PART 1: Setting Up the Main Tasks
(c) PART 2: Synthesizing the Selector
(d) PART 3: Synthesizing the Constructor
(e) PART 4: Completing the Program

TYPES OF PROGRAMMING KNOWLEDGE

SUMMARY AND CONCLUSIONS

ACKNOWLEDGMENTS

REFERENCES

INTRODUCTION

Summary

The overall objective of our research is to gain more insight into the pro-

gramming process, as a necessary step toward building *program-understanding systems*. Our approach has been to examine the process of synthesizing very simple programs in the domain of sorting. We hope that by beginning with this simple domain and developing and implementing a reasonably comprehensive theory, we can then gauge what is required to create more powerful and general program-understanding systems.

Toward this end, we are working on first isolating and codifying the knowledge appropriate for the synthesis and understanding of programs in this class and then embedding this knowledge as a set of rules in a computer program. Along the way, we have developed some preliminary views about what a program-understanding system should know.

Our goal in this particular paper is to present a *dialogue* with a *hypothetical* program-understanding system. A dialogue was chosen as a method of presentation that would exemplify, in an easily understood fashion, what such as system should know. The subject of the dialogue is the synthesis of a simple insertion sort program. Each step in the dialogue corresponds to the utilization of one or more pieces of suggested programming knowledge. Most of this knowledge is stated explicitly in each step. The dialogue presented here is a highly fictional one, although some portions of the reasoning shown in the dialogue have been tested in an experimental system.

We are now in the process of formulating the necessary programming knowledge as a set of synthesis rules. However, the scope of this paper does not include the presentation of the current state of our rules. So far some 150 rules have been developed and debugged in a rule-testing system. The synthesis tasks on which these rules are being debugged include two insertion sorts, one selection sort, and a list reversal. In another paper we present a description of the rules for synthesis of an insertion sort program (Green and Barstow, 1975). The methods presented here have also led to the implementation of a "coding" module as part of the more comprehensive "PSI" program synthesis system. PSI is further discussed in an overview of the entire system (Green, 1976), and in a description of the coding module and the related efficiency module (Barstow and Kant, 1976).

As will become apparent in the dialogue, one of our conjectures is that a program-understanding system will need very large amounts of many different kinds of knowledge. This seems to be the key to the flexibility necessary to synthesize, analyze, modify, and debug a large class of programs. In addition to the usual types of programming knowledge, such as the semantics of programming languages or techniques of local optimization, many other types are needed. These include, at least, high-level programming constructs, strategy or planning information, domain-specific and general programming knowledge, and global optimization techniques. In Section III we discuss this further and show where these kinds of knowledge appear in the dialogue.

Domain of discourse

Topics mentioned in the dialogue include data structures, low-level opera-

tions, and high-level programming constructs. The main data structures mentioned in our dialogue are ordered sets represented by linked lists. The low-level operations mentioned include assignment, pointer manipulation, list insertion, etc. Some of the higher-level notions or constructs we consider are permutation, ordering (by various criteria), set enumeration, generate and test, generate and process, proof by induction, conservation of elements during a transfer, and methods of temporary marking (or place-saving) of positions and elements. Time and space requirements for various methods are not discussed.

The target language is LISP, in particular the INTERLISP language (Teitelman, *et al.*, 1974). However, in the dialogue we represent the programs in a fictitious meta-LISP.

A DIALOGUE

Introduction

In this section we wish to exhibit what we consider to be a reasonable level of understanding on the part of a program-understanding system. It is not obvious how best to present this in a way that is easy for the reader to follow, since the synthesis process is rather complex. We hope that an English language dialogue is adequate. We have added to the English several "snapshots" of the developing program that help to indicate where the system is in the programming process. These diagrams show a structure similar to the stepwise refinements used in structured programming (Dahl, Dijkstra, and Hoare, 1972). Our dialogue may be considered as a continuation of the technique of presentation used by Floyd for a program verifier-synthesizer (Floyd, 1971), although our more hypothetical system has been allowed to know more about program synthesis for its domain of discourse.

In certain ways we feel that the dialogue is *not* representative of how a program-understanding system would appear to the user during the synthesis process (although such a low-level dialogue would at times have its place). For expository purposes the dialogue has certain choices and inferences made by the machine and others made by the user. Depending upon the application, these decisions and inferences could reasonably be redistributed between user and machine, with some made by other automated "experts", such as an efficiency expert, and other decisions forced by the context in which the program is written. For example, the data structures for this insertion sort might be determined in advance if the sort routine were part of a larger system, and all choices made by the user could be made instead by rules calculating the efficiency of each branch. A more typical user-machine dialogue would usually be conducted at a much higher level and in the words used to talk about the application area for the program; but to present a hypothetical dialogue in which the program merely said, "OK, I'm all finished," or, "I wrote a sort program as part of your operating system," wouldn't explicate very much programming knowledge. Some of these ideas about choice-making by an efficiency expert and specifying programs through high-level dialogue have been embodied in the PSI program

synthesis system.

The style of the dialogue is not intended to be a strong suggestion about what the process of *specifying* a program should be. In fact, we feel that program specification is a difficult problem in automatic programming research (Green, *et al.*, 1974), and we touch on only a few aspects of it here. However, some of the techniques used in the dialogue will probably find some, as yet undetermined, place in program-understanding systems. The obvious techniques include interactive specification, natural language communication, and the periodic presentation of summaries of the structure and content of the evolving program.

We note that we have endowed our hypothetical system with various abilities, including an inference capability, a program simplification capability, and the ability to illustrate and summarize where it is in the programming process. Further, the dialogue sometimes takes on the character of a decision-tree, sometimes of goal-oriented problem-solving, and sometimes of stepwise refinement. These all seem appropriate to the task.

Various decisions, such as how large to make a step, when to suppress or explicate an inference, and when to present a diagram, were made somewhat arbitrarily, but primarily on the basis of what seemed to be the clearest exposition.

Before the start of the dialogue, we wish to remind the user that the focus is not on the particular mechanisms which might be employed to produce the hypothesized behavior, but rather on the reasoning itself.

Part 1: Setting up the main tasks

1. **User:** Please write me a *sort* program. Use *linked lists* to represent the input and output, which are ordered sets. The output set should be ordered so that each element is *numerically greater than* all preceding elements. The input set will be called **INPUT** and the output set will be called **OUTPUT**.
2. **Machine:** As I understand it, a sort program produces as output a permutation (of the input) that satisfies a test. In this case, the test is that the elements be ordered on the basis of the above-mentioned ordering relation, "numerically greater than". Please choose one of the two known methods† for finding a permutation that satisifies a test:
   (a) Generate and test permutations until the desired one is found.
   (b) Carefully construct the desired permutation.
3. **User:** Carefully construct the desired permutation.
4. **Machine:** Should the output set be constructed in the same list cells as the input set, or should a new list be created?
5. **User:** A new list should be created.‡

†The limitation to only two "known" choices is arbitrary, but it is reasonable to expect a system to have a limited number of choices.

‡This does not preclude overwriting the input set during the sort. Overwriting might be done in order to indicate which elements have been selected from the input.

6. **Machine**: Do you want an iterative or recursive program?
7. **User**: Iterative. [We wish to remind the reader that the distinctions between "user" and "machine" are somewhat arbitrary. Yet a decision must be made, so we have chosen to let our hypothetical "user" make it. Furthermore there is no reason to expect the user to have a preference for an iterative or recursive program. We have our user choose an iterative structure over what might be a more natural recursive structure, because there are fundamental aspects of the algorithm which are more explicit in its iterative form.]
8. **Machine**: Then the program will be in the form of a *transfer program* which must be *total*§. The program will transfer all elements from the input to positions in the output, such that the final output is a permutation (of the input) which satisfies the above ordering relation, namely increasing by numerical value.

The transfer paradigm†‡

```
transfer program:
   LOOP UNTIL all elements of input have been processed;
        selector:
                select the next element from input;
        constructor:
                add this element to output;
   REPEAT.
```

9. **Machine**: In order to write such a transfer program, I must achieve several goals:

   (1) Write a *selector*. On each call, the selector will find that element of the input which comes next in the chosen transfer order (see 3 below). The selector consists of two parts:

   (a) selector initializer
   (b) selector body

   (2) Write a *constructor*. On each call, the constructor will produce a new output (or modify the old output), so that the new element is in the appropriate position but that otherwise the

§By "total", we mean that the transfer loop must be repeated until all elements have been transferred.

†As illustrated, the transfer program has a "selector" part and a "constructor" part. The selector produces the elements one at a time, and the constructor builds the new set. Either the selector or the constructor may re-order the set, i.e., the selector can produce the elements in an order different from the stored order, or the constructor can build a set with an ordering different from the order of receipt of the elements.

‡We will generally present in our diagrams only those parts which seem relevant to the current context.

order of elements in the output is unchanged. The constructor consists of two parts:

(a) constructor initializer
(b) constructor body

Before I can write these parts, I must:

(3) Select a *transfer order*–the order in which the variables will be transferred from the input to the output.

(4) Select a *transfer termination method*–a test that all elements have been transferred, and a position in the program appropriate to the type of test selected.

(5) Select a name for the *transfer variable*–a variable which holds as its value the element being transferred.

After writing the parts, I must:

(6) Combine the selector, constructor and transfer termination test into a *generate and process* loop. This completes the final program.

The basic structure of the program (including test position)†

```
transfer initializer:
    selector initializer:                                    (1a)
    constructor initializer:                                 (2a)
transfer body:
    LOOP UNTIL DONE;
        transfer termination test:
            IF all elements have been transferred            (4)
                THEN DONE;
        selector body:
            select the next element from input;              (1b)
        constructor body:
            add the element to output;                       (2b)
    REPEAT;
    DONE:    HALT.
```

10. Machine: Should the transfer order be *linear* with respect to the intial order or with respect to the final order?‡

†Our syntax for iterative statements and exits is that proposed by Zahn (Zahn, 1974) and supported by Knuth (Knuth, 1974). Loops are exited upon encountering an *event*, such as **DONE** in the above example. The program exits to the occurrence of the event label after the body of the loop. Events are declared after *LOOP UNTIL*. *REPEAT* denotes the end of the loop.

‡We define the "initial" and "final" orders to be the conventional list-representation ordering of the input set and of the desired output set, respectively. By "linear with respect to an order" we mean either identical to that order or the converse of that order. The user's choice of "linear with respect to the intial order" amounts to choosing an insertion sort algorithm, rather than a selection sort algorithm. The full effect of this choice will be seen when choosing the "construction re-ordering" during the synthesis of the constructor.

11. User: Linear with respect to the initial order.
12. Machine: Should the transfer order be identical to the initial order or the converse of the initial order?
13. User: Identical.
14. Machine: Now let's select a transfer termination method. This will consist of a test and a position for the test. Since the transfer will be total, the transfer termination test must test whether all elements have been transferred. This is equivalent to testing whether no elements remain to be selected. This is the same test which is used by the selector to stop its generation of elements. I will use this test as the transfer termination test. While writing the selector I will thus have to write a third part: the selector termination test. Since this test must precede a call to the selector body, I will put the test at the beginning of the loop.†
15. Machine: I will use **ELEMENT** as the transfer variable.‡

Part 2: Synthesizing the selector

We are about to work on the indicated§ parts of the selector

```
transfer initializer:
        selector initializer:                                    *
        constructor initializer:
transfer body;
        LOOP UNTIL DONE;
                transfer termination test:
                        IF all elements have been transferred⇒
                                all element have been selected   *
                                THEN DONE;
                selector body:
                        select the next element from input;      *
                constructor body:
                        add the element to output;
        REPEAT;
        DONE:  HALT.
```

†For ease in presentation the preceding diagram showed the result of this choice (i.e., the test appeared in the proper location), although the choice had not yet been made. Considering all of the tests which might have been used and all of the places where such a test might have been put, one could imagine a great deal of planning going into making this rather difficult decision.

‡To choose meaningful variable names requires knowledge about communication with the user and is a difficult problem.

§Throughout this dialogue, we will use special characters and type fonts to indicate different sections of these diagrams. We present the meaning of the fonts here in case confusion arises, but the meaning should be clear from the context.

areas not under consideration
    [with • • • indicating areas omitted]
areas currently under consideration
PROGRAM PARTS JUST COMPLETED
RESERVED WORDS
BASIC LISP FUNCTIONS
USER- OR MACHINE-DEFINED LABELS
⇒ the right arrow indicates a step in the refinement process

16. **Machine:** I will now choose a *selection re-ordering.* I will use the term "selection re-ordering" to refer to the relation between (a) the order of the elements in the input (positional order) and (b) the order in which the elements are selected by the selector (temporal order). Since the transfer order is *linear* and *forward* (that is, identical to the input order), the selection re-ordering must also be linear and forward.
17. **Machine:** A selector is a type of *generator*, a "total" generator that produces each element of the input once and only once. Corresponding to the three parts of a selector, this type of generator also has three parts:‡
    (1) a body which actually produces the elements
    (2) an initializer which can set up the generator before it is called the first time
    (3) a termination test (to be used as the transfer termination test) which tests whether all elements have been generated
18. **Machine:** I must now select the *state-saving scheme* by which the generator will save its state after each call. In order to save its state, the generator must remember how the input set is divided into two sets:
    (1) those elements which have already been generated
    (2) those elements which are yet to be generated

    Recall that the generator is moving linearly through the input list from first to last to find each new element. Therefore all elements before its current position in the list have already been generated, and all elements after its current position are yet to be generated. To save the state of the generator, I need only save a marker for the generator's current position in the input list. One pointer will be adequate for this purpose. Let us call this pointer **REST.**

We are about to work on the indicated parts of the selector body

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE;
   • • •
         selector body:
               select the next element from input⇒
                     generate the next element from input⇒
                           produce element;                    *(2)
                           increment state of generator;       *(1)
   • • •
   REPEAT;
   DONE:  HALT.
```

‡Technically, before these parts can be written, the production re-ordering for the generator must be chosen. In this case, it is obviously the same as the selection re-ordering, so we have omitted it from the dialogue in the interests of clarity. The initializer is written after the producer, since in general the form of an initializer is dependent on the body of the loop to be initialized.

19. Machine: Now I will synthesize the body of the generator. I need code for two pieces:
    (1) increment the state of the generator
    (2) produce the next element

20. Machine: To increment the state of the generator, I will advance the position pointer by one. Since the representation is a linked list, the code for this is merely: REST ← *CDR*(REST)†

21. Machine: Now I will write the producer. Since the state-saving scheme uses a pointer, **REST**, to separate those elements generated from those not yet generated, and since the next element to be generated is the first element of the non-generated set, then the process of saving the state will also effectively mark the next element to be generated.‡ To produce the next element, then, I need merely take the first element of the non-generated set, which is pointed to by **REST**.§ Since the representation is a linked list, the subprogram for this is: ELEMENT ← *CAR*(REST)

22. Machine: I now put the pieces of the generator body together into a block:

ELEMENT ← *CAR*(REST);
REST ← *CDR*(REST)

We have just finished the indicated parts

```
transfer initializer:
   . . .
transfer body:
   LOOP UNTIL DONE;
   . . .
         selector body:
               select the next element from input⇒
                     generate the next element from input⇒
                           produce element⇒
                                 ELEMENT ← CAR(REST);          ★
                           increment state of generator⇒
                                 REST ← CDR(REST);             ★
   . . .
   REPEAT;
   DONE:  HALT.
```

†Throughout, we will use the standard LISP terminology:
CAR(L) gives the first element of L
CDR(L) gives the remaining elements of L
CONS(X L) creates a new list with the first element X and with the rest of the elements being those of the list L
NIL is the empty list (and list terminator)

‡In a selection sort, the code to find the next element to be generated would be considerably more complex since it would need to find the largest of the remaining elements.

§There are situations in which the value of the element or a pointer into the list might be returned, rather than the element itself.

23. **Machine:** I will now write the generator initializer and the generator termination test. Since I am using the pointer **REST** to mark the elements not generated, and since initially no elements have been generated, I must initialize it to the entire input set:

REST ← INPUT

24. **Machine:** Now let's write the generator termination test (which will be used as the transfer termination test). To test whether all elements have been generated, I will test whether there are no elements remaining. Since the elements are being generated in a forward linear order, I simply test for being at the end of the input list. Since the representation is a linked list, this is simply a test for emptiness:

REST = *NIL* ?

We have just finished the indicated parts

```
transfer initializer:
    selector initializer:➡
        generator initializer:➡
            REST ← INPUT;                                    ★
    • • •
transfer body:
    LOOP UNTIL DONE;
        transfer termination test:
            IF all elements have been transferred➡
                all elements have been selected➡
                    all elements have geen generated➡
                        no elements remain➡
                            REST = NIL                       ★
                THEN DONE;
        selector body:
            select the next element from input➡
                generate the next element from input ➡
                    produce element➡
                        ELEMENT ← CAR(REST);
                    increment state of generator➡
                        REST ← CDR(REST);
    • • •
    REPEAT;
    DONE:  HALT.
```

25. **Machine:** I have finally finished the generator which the selector needed. This completes the selector.

Part 3: Synthesizing the constructor

We are now working on the indicated parts of the constructor

```
transfer initializer:
    selector initializer:
    constructor initializer:                                    *
transfer body:
    LOOP UNTIL DONE;
        transfer termination test:
            IF all elements have been transferred→
                all elements have been selected
                THEN DONE;
        selector body:
            select the next element from input;
        constructor body:
            add the element to output;                          *
    REPEAT;
    DONE:  HALT.
```

26. **Machine:** One way to create an output set that is a permutation of the elements received by the constructor is to write an *inductive* constructor that satisfies the following constraints:

    (1) The initial output set must be empty.
    (2) Each construction operation must be such that the new output set consists of the old set plus one new element.†
    (3) After each construction operation, the new set must satisfy our ordering relation.

27. **Machine:** To write such a constructor, two decisions must first be made:

    (1) Choose a construction re-ordering.
    (2) Choose a construction method for building a new set, given the old set, the new element, and the position where the new element should occur in the set.

    After these choices are made, we will produce code for the constructor body and then the constructor initializer.‡

28. **Machine:** I will now choose a *construction re-ordering*. I will use the term "construction re-ordering" to refer to the relation between (a) the order in which the elements are received by the constructor (temporal order) and (b) the order of the elements in the output (positonal order). The re-

†It is sometimes convenient to think of the constructor as producing a sequence of output sets, and sometimes as successively modifying one output set.

‡Note that we have implicitly decided that the construction consists of two steps: (1) finding the position; (2) adding the element to the set. One might imagine schemes in which finding a position and modifying the set are interleaved operations.

orderings may be linear or complex, where linear means same as or converse of. Since the order in which the elements are received (the transfer order) is the same as the input order, and the desired output order is not necessarily the same as (nor the converse of) the input order, the construction re-ordering must be complex (non-linear).† This re-ordering will be based on the "numerically greater than" ordering relation. This means that the correct position of each element in the output list will be such that all preceding elements are less than or equal to the new element and all following elements are greater than the new element.‡

29. **Machine:** Now we must select a *construction method* for building a new set from the old one. The choices are:

    (a) List insertion: Modify the structure of the old list by inserting a new cell containing the new element.

    (b) Shifting: Allocate a new space at the beginning (or end) of the set and shift all elements before (after) the new element by one position. Then put the new element into the vacant place.§

    (c) Copying: Copy the old set into a new space, one cell larger than the old space, with the new element taking up the extra position.

    Which shall I use?

30. **User:** List insertion.

31. **Machine:** Now I will write the body of the constructor. The body consists of two parts:

    (1) a *position finder*, which finds the position where the new element belongs

    (2) a *list inserter*, which inserts the element at this position

32. **Machine:** When a position has been found, there are several ways to indicate that position. First we must select one of these representations for a position. We are considering positions to be between two elements in a list. Since the construction method involves inserting a new cell into the list, we must choose a representation appropriate to insertions into linked lists. I know about two types of such representations:

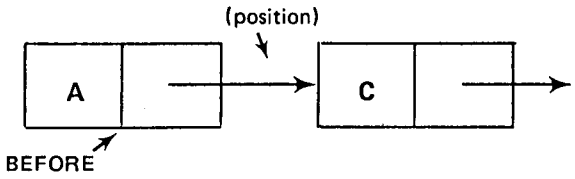
---

†This choice of a complex construction re-ordering, along with the choice of a linear selection re-ordering, completes the specification of an insertion sort, all of which follow from the user's choice of a transfer order "linear with respect to the initial order".

‡For clarity, we will refer throughout to an element being "less than" or "greater than" another element and mean that the first element precedes or follows the second element according to the chosen ordering relation.

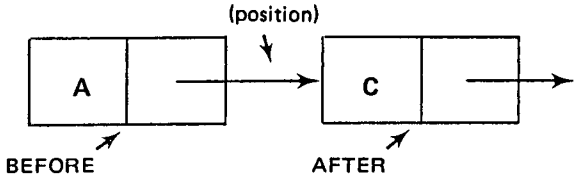
§Perhaps, since the "shifting" operation is such an unlikely choice when lists are being used, it shouldn't be presented to the user. It could be called forth only upon questioning or in unusual programming contexts.

(a) Use one pointer as illustrated:



In using this representation, a special unique value for BEFORE must be used if the correct position is at the front of the list. "FRONT" will be used for this purpose.

(b) Use two pointers as illustrated:



In using this representation, a special unique value for BEFORE must be used if the correct position is at the front of the list. "FRONT" will be used for this purpose. If the correct position is at the end of the list, AFTER will point to the empty list.

Which shall I use?

33. User: Please use two pointers. [This choice might be based either on looking ahead or on knowing at this point that the BEFORE pointer is useful for inserting into the list and the AFTER pointer is useful for incrementing the state of the position finder and for testing the correctness of a position.]

34. Machine: I need variable names for the two pointers. I will use **BEFORE** and **AFTER**.

We are about to work on the position finder as indicated

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE;
         • • •
         constructor body:
               add the element to output⇒
                     find the position for the new element;          *
                     insert the new element into the position;
   REPEAT;
   DONE:  HALT.
```

35. Machine: I am now writing the program that finds the position. First we will select a *search strategy* for searching the output list to find the correct

position. The only strategy which I know about for linked lists is a *forward linear scan*, so I will use that strategy.

36. **Machine:** Since the search strategy is a forward linear scan of a finite list, the search strategy is total; that is, all possible positions will be considered. Since the old output set is well-ordered, and since the ordering relation applies to all elements involved, one of these possible positions must be correct. Therefore, the search for a correct position must succeed, and no exit will be required for the case of not finding a correct position. So I will use a loop which consists only of a test and exit on the correctness of the position and an incrementer. The incrementer will increment the state of the search according to the search strategy and specify the next position to be considered. I will also need to initialize the loop.

37. **Machine:** Since the search strategy is forward linear, the correctness test and the incrementer are independent. This means that the next position specified by the incrementer does not depend on the results of the correctness test. This allows a simple non-branching incrementer.†

We are about to work on the indicated parts of the position finder

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE;
      • • •
      constructor body:
         add the element to output⇒
            find the position for the new element⇒
               initialize position finder loop;                    *(2)
               LOOP UNTIL CORRECT POSITION FOUND;
                  IF position is correct                           *(3)
                     THEN CORRECT POSITION FOUND;
                  increment state and specify position;            *(1)
               REPEAT;
               CORRECT POSITION FOUND;
            insert the new element into the position;
   REPEAT;
   DONE:  HALT.
```

38. **Machine:** Now let's design the incrementer. First we must select a scheme for representing the state of the search. The incrementer will increment this state and will also specify the next position to be considered.

39. **Machine:** Since the search strategy is a foward linear scan of the linked list, and since the two-pointer position representation has been selected, the

†If the search strategy had been to use a binary chop, for example, then the next position to be tested would be based on the results of the test.

two pointers (**BEFORE** and **AFTER**), which are already being used to represent the position, are also sufficient to remember the state.†

40. **Machine:** An incrementer for a forward linear scan using the two-pointer position representation requires two pieces of code: one to reset **BEFORE** to the value of **AFTER**, and one to increment **AFTER**. This is adequate both for the general case in the interior of a linked list, and in the special case at the front of a list, where **BEFORE** originally has the special value **"FRONT"** and is reset to point to the first cell in the list. The pieces will be put together in that order.
41. **Machine:** I am now writing the code to reset **BEFORE** to the value of **AFTER**. The code to do that is:

BEFORE ← AFTER

42. **Machine:** I am now writing the code to increment **AFTER**. Since the representation is a linked list, the code to do that is:

AFTER ← *CDR*(AFTER)

43. **Machine:** Putting these two pieces together, we have the following incrementer for the position finder:

BEFORE ← AFTER;
AFTER ← *CDR*(AFTER)

44. **Machine:** I will also need a position initializer which initializes both the state of the position finder and the position representation to the first position according to the search strategy.
45. **Machine:** Since the position representation is also used to represent the state, we need only initialize the position representation.
46. **Machine:** Since the search strategy is forward linear scan, we must initialize the position representation to the position at the front of the list. The convention for representing the position at the front of the list is that **BEFORE** has the unique value **"FRONT"** and that **AFTER** points to the entire list.
47. **Machine:** The code to set **BEFORE** to **"FRONT"** is:

BEFORE ← "FRONT"

48. **Machine:** The code to set **AFTER** to the entire list is:

AFTER ← OUTPUT

---

†The position memory and the state memory need not always be the same. For example, if only **BEFORE** were used to remember the position, a good choice of state-memory would probably include a pointer playing the same role as **AFTER**.

49. **Machine:** Putting these two pieces together, the initializer for the position finder is:

```
BEFORE ← "FRONT";
AFTER ← OUTPUT
```

We have just finished the indicated parts of the position finder

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE;
      • • •
      constructor body:
         add the element to output⇒
            find the position for the new element⇒
               initialize position finder loop⇒
                  BEFORE ← "FRONT";                    ★
                  AFTER ← OUTPUT;                      ★
               LOOP UNTIL CORRECT POSITION FOUND;
                  IF position is correct
                     THEN CORRECT POSITION FOUND;
                  increment state and specify position⇒
                     BEFORE ← AFTER;                   ★
                     AFTER ← CDR(AFTER);               ★
               REPEAT;
               CORRECT POSITION FOUND:
            insert the new element into the position;
   REPEAT;
   DONE:  HALT.
```

50. **Machine:** I will now work on the exit test for the correct position. A position is correct if the new element is greater than all elements preceding the position and less than all elements following the position. I will first try to infer the necessary pieces of this test.
51. **Machine:** Since the set is ordered, it follows from transitivity that the new element need not be compared against all of the elements of the set, but merely against the elements immediately adjacent to the position being considered.
52. **Machine:** Since the search strategy is forward linear scan, at any point in the search, all of the elements occurring before the current position in the list have already been tested and found to be less than the new element. This holds for all positions including the initial position in which no elements occur before the position in the list.
53. **Machine:** Thus, the test for correctness of position need only include a test that all elements occurring after the current position in the list are greater than the new element.
54. **Machine:** To test that all elements occurring after the current position are

greater than the new element, the program must first test whether there are no elements occurring after the current position. Then, if there are such elements, it must test whether all such elements are greater than the new element.

55. **Machine:** A test that there are no elements occurring after the current position is simply a test whether the list pointed to by the second pointer **(AFTER)** is empty:

AFTER = *NIL* ?

56. **Machine:** As noted previously, the correctness test need only consider the immediately adjacent elements, so that a test that all elements occurring after the current position are greater than the new element is simply a test that the first element occurring after the current position is greater than the new element. To write such a test I need code to extract the first element from the list pointed to by **AFTER** and a test whether this element is greater than the new element.

57. **Machine:** The code to extract the first element from a list is:

CAR(AFTER)

58. **Machine:** The code to test whether this element is greater than the new element is:

CAR(AFTER) > ELEMENT ?

59. **Machine:** Given the two partial tests, the complete correctness test is an ordered *OR*† of the two partial tests:

[AFTER = *NIL*] *OR* [*CAR*(AFTER) > ELEMENT] ?

†By "ordered *OR*" we mean that the second argument is evaluated only if the first argument is false.

The indicated parts complete the position finder

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE;
      • • •
      constructor body:
         add the element to output⇒
            find the position for the new element⇒
               initialize position finder loop⇒
                  BEFORE ← "FRONT";
                  AFTER ← OUTPUT;
               LOOP UNTIL CORRECT POSITION FOUND;
                  IF position is correct⇒
                     no elements occur after position⇒
                        AFTER = NIL                          ★
                     OR next element is greater⇒
                        CAR(AFTER) > ELEMENT                 ★
                     THEN CORRECT POSITION FOUND;
                  increment state and specify position⇒
                     BEFORE ← AFTER;
                     AFTER ← CDR(AFTER);
               REPEAT;
               CORRECT POSITION FOUND;
            insert the new element into the position;
   REPEAT;
   DONE:  HALT.
```

60. Machine: Having completed the position finder, we must now write the list inserter which takes a new element, a position represented by the two-pointer scheme, and the output set, and produces a new output set with the new element in the indicated position. The method of inserting an element into a list depends on whether or not the indicated position is at the front of the list. I will need to write a test to separate the "front of the list" case from the "anywhere else" case, and use a different method for each case.

We are about to work on the indicated parts of the list inserter

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE;
      • • •
      constructor body:
         add the element to output⇒
            find the position for the new element;
            insert the new element into the position⇒
               IF position is at front                       ★
                  THEN add element onto front                ★
                  ELSE insert into interior                  ★
   REPEAT;
   DONE:  HALT.
```

61. Machine: A test for being at the front of the list is a test whether the value of the first pointer is **"FRONT"**. The code for this is:

BEFORE = "FRONT" ?

62. Machine: The addition of the new element in the "front of the list" case entails no replacement into the list, but instead an addition to the front of the list. The code to do this is:

OUTPUT ← *CONS*(ELEMENT OUTPUT)

63. Machine: The addition of the new element anywhere else in the list requires a replacement into a cell in the list. The code to do this is:

CDR(BEFORE) ← *CONS*(ELEMENT AFTER)†

The marked parts complete the element adder

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE;
      • • •
      constructor body:
         add the element to output⇒
            find the position for the new element;
            insert the new element into the position⇒
               IF position is at front⇒
                  BEFORE = "FRONT"                          ★
                  THEN add element onto front⇒
                     OUTPUT ← CONS(ELEMENT OUTPUT)          ★
                  ELSE insert into interior⇒
                     CDR(BEFORE) ← CONS(ELEMENT AFTER);     ★
   REPEAT;
   DONE:  HALT.
```

64. Machine: Having completed the list inserter (and thus the constructor body), we must now write the constructor initializer. Since we are writing an inductive constructor that must begin with an empty output list, we simply set the output list to the empty list:

OUTPUT ← *NIL*

---

† LISP programmers will recognize this as *RPLACD*.

The initializer completes the constructor

```
transfer initializer:
   • • •
   constructor initializer⇒
      OUTPUT ← NIL;                                          ★
transfer body:
   LOOP UNTIL DONE;
      • • •
      constructor body:
         add the element to output⇒
            find the position for the new element⇒
               initialize position finder loop⇒
                  BEFORE ← "FRONT";
                  AFTER ← OUTPUT;
               LOOP UNTIL CORRECT POSITION FOUND;
                  IF position is correct⇒
                     no elements occur after position⇒
                        AFTER = NIL
                     OR next element is greater⇒
                        CAR(AFTER) > ELEMENT
                     THEN CORRECT POSITION FOUND;
                  increment state and specify position⇒
                     BEFORE ← AFTER;
                     AFTER ← CDR(AFTER);
               REPEAT;
               CORRECT POSITION FOUND:
            insert the new element into the position⇒
               IF position is at front⇒
                  BEFORE = "FRONT"
                  THEN add element onto front⇒
                     OUTPUT ← CONS(ELEMENT OUTPUT)
                  ELSE insert into interior⇒
                     CDR(BEFORE) ← CONS(ELEMENT AFTER);
   REPEAT;
   DONE:  HALT.
```

65. Machine: We have now finished the constructor.

Part 4: Completing the program

The final program

```
transfer initializer:
    selector initializer:=>
        generator initializer:=>
            REST ← INPUT;
    constructor initializer=>
        OUTPUT ← NIL;
transfer body:
    LOOP UNTIL DONE;
        transfer termination test:
            IF all elements have been transferred=>
                all elements have been selected=>
                    all elements have been generated=>
                        no elements remain=>
                            REST = NIL
                THEN DONE;
        selector body:
            select the next element from input=>
                generate the next element from input=>
                    produce element=>
                        ELEMENT ← CAR(REST);
                    increment state of generator =>
                        REST ← CDR(REST);
        constructor body:
            add the element to output=>
                find the position for the new element=>
                    initialize position finder loop=>
                        BEFORE ← "FRONT";
                        AFTER ← OUTPUT;
                    LOOP UNTIL CORRECT POSITION FOUND;
                        IF position is correct=>
                            no elements occur after position=>
                                AFTER = NIL
                            OR next element is greater=>
                                CAR(AFTER) > ELEMENT
                            THEN CORRECT POSITION FOUND;
                        increment state and specify position=>
                            BEFORE ← AFTER;
                            AFTER ← CDR(AFTER);
                    REPEAT;
                    CORRECT POSITION FOUND:
                insert the new element into the position=>
                    IF position is at front=>
                        BEFORE = "FRONT"
                        THEN add element onto front=>
                            OUTPUT ← CONS(ELEMENT OUTPUT)
                        ELSE insert into interior=>
                            CDR(BEFORE) ← CONS(ELEMENT AFTER);
    REPEAT;
    DONE:  HALT.
```

66. Machine: Having completed the pieces of the selector and the pieces of the constructor, I can assemble the pieces into the final program using a generate and process paradigm with the termination test before the generator.

67. Machine: The final program, without any meta-code or comments, and its translation into LISP are given below:

```
REST←INPUT;
OUTPUT←NIL;
LOOP UNTIL DONE;
 IF
  REST=NIL
  THEN DONE;
 ELEMENT←CAR(REST);
 REST←CDR(REST);
 BEFORE←"FRONT";
 AFTER←OUTPUT;
 LOOP UNTIL CORRECT POSITION FOUND;
  IF
   AFTER=NIL
   OR CAR(AFTER))ELEMENT

   THEN CORRECT POSITION FOUND;
  BEFORE←AFTER;
  AFTER←CDR(AFTER);
 REPEAT;
 CORRECT POSITION FOUND:
 IF
  BEFORE="FRONT"
  THEN
   OUTPUT←
    CONS(ELEMENT OUTPUT)
  ELSE
   CDR(BEFORE)←
    CONS(ELEMENT AFTER);
REPEAT;
DONE: HALT.
```

```
[LAMBDA NIL
 (PROG NIL
  (SETQ REST INPUT)
  (SETQ OUTPUT NIL)
 L1
  (COND
   ((NULL REST)
     (GO L2)))
  (SETQ ELEMENT (CAR REST))
  (SETQ REST (CDR REST))
  (SETQ BEFORE "FRONT")
  (SETQ AFTER OUTPUT)
 L3
  (COND
   ((OR (NULL AFTER)
        (GREATERP (CAR AFTER)
                  ELEMENT))
     (GO L4)))
  (SETQ BEFORE AFTER)
  (SETQ AFTER (CDR AFTER))
  (GO L3)
 L4
  (COND
   ((EQUAL BEFORE "FRONT")

     (SETQ OUTPUT
       (CONS ELEMENT OUTPUT)))
   (T
     (RPLACD BEFORE
       (CONS ELEMENT AFTER))))
  (GO L1)
 L2
  (RETURN NIL]
```

TYPES OF PROGRAMMING KNOWLEDGE

On reviewing the diálogue, we can see that there are several types of knowledge involved. We first note that there is significant use of a kind of strategy or planning knowledge. On one level, we see this in steps 9 and 14, where the system discusses what must be done to write a transfer program. In step 9 for example, the sub-steps 3 and 4, where the transfer order and the transfer termination method are chosen, are really a kind of strategy for determining the form that the basic algorithm will take. On a different level, we see a kind of global

optimization in steps 21 and 39, where the system decides that information structures designed for one purpose are sufficient for another. In step 21, for example, the pointer originally chosen to save the state of the selector (by marking the dividing point between those elements generated and those not yet generated) is found to be adequate for the purpose of indicating the next element to be generated. One could imagine, as an alternative to this type of planning, the use of more conventional local optimization such as post-synthesis removal or combination of redundant portions.

We also see that the system makes considerable use of inference and simplification knowledge. Inference plays a role in the global optimization planning mentioned above, and also appears in steps 16 and 28, where the selection and construction re-orderings are determined. Simplification and inference are both apparent in steps 49 through 56, where the test for the correctness of the position is reduced to a simple test on the variable **AFTER**. Simplification and inference are also needed in step 36 where the system decides that an error exit (for the case of no position being found) is unnecessary.

Additionally, there are types of knowledge which are spread throughout the dialogue. Relatively domain-specific knowledge (in this case, about sorting) is particularly necessary in the earlier stages. Language-specific knowledge (in this case, about LISP) is necessary when the final code is being generated. General programming knowledge, such as knowledge about set enumeration and linked lists, is necessary throughout the synthesis process. Further, one could imagine significant use of efficiency information, although it is not present in our particular dialogue.

The variety of types and amounts of knowledge used in the dialogue would tend to indicate that much more information is required for automatic synthesis of sorting programs than appeared in earlier, computer-implemented, systems for writing sort programs (Green, 1969; Kowalski, 1974; van Emden, 1974). Ruth has developed a formulation of the knowledge involved in interchange and bubble sort programs (Ruth, 1974). His formulation is aimed primarily at the analysis of simple student programs in an instructional environment and the analysis task as defined does not seem to require the same depth and generality of knowledge suggested by our dialogue. Our intuition is that a significantly greater depth of programming knowledge would be required to extend his formulation to a larger class of programs. It is also interesting to compare the information involved in our dialogue to that found in non-implemented (and not intended for machine implementation) human-oriented guides for sort-algorithm selection and in textbooks on sorting. Martin (Martin, 1971) gives methods for selecting a good algorithm for a particular sorting problem. Those algorithms are much more powerful than those we deal with and their derivation would require considerably more information. Wenote that at the level of algorithm description presented, little explicit information is available to allow pieces of algorithms to be fitted together or to allow slight modification of existing algorithms. A sorting textbook such as (Knuth, 1973), gives several orders of magnitude more

information on sorting than is required for our dialogue.

Can we measure or estimate in some way how much knowledge is necessary for program-understanding systems? The fact that the dialogue describing the synthesis took some seventy steps (with some of the steps rather complex) is an indication that considerable information is involved. In our experiments, we have found that about one hundred fifty explicitly stated "facts" or rules would get a synthesis system through the underlying steps of this dialogue. Furthermore, it is our guess that at least this much knowledge density will be required for other similar tasks, in order to have the flexibility necessary for the many aspects of program understanding. Although we are suggesting that such information must be effectively available in some form to a system, we are not in a position to estimate how much of this information should be stated explicitly (as, say, rules), how much should be derivable (from, say, meta-rules), how much should be learned from experience, or available in any other fashion.

SUMMARY AND CONCLUSIONS

In this paper we have tried to exemplify and specify the knowledge appropriate for a program-understanding system which can synthesize small programs, by presenting a dialogue between a hypothetical version of such a system and a user. Our conjecture is that unless a system is capable of exceeding the reasoning power, and even some of the communication abilities, exemplified by the dialogue, the system will not effectively "understand" what it is doing well enough to synthesize, analyze, modify, and debug programs. It appears that a system which attempts to meet this standard must have large amounts of many different kinds of knowledge. Most such programming knowledge remains to be codified into some form of machine implementable theory. In fact, the codification of such knowledge is one of the main research problems in program-understanding systems.

As for our own work, in the near future we expect to extend the rules to deal with several different types of sorting programs and set operations. Perhaps then we will be in a better position to estimate the requirements of larger program-understanding systems.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the helpful suggestions given by Avra J. Cohn, Brian P. McCune, Richard J. Waldinger, and Elaine Kant after numerous readings of earlier drafts of this paper. Computer time for our preliminary tests was made available by the Artificial Intelligence Center of the Stanford Research Institute. This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the U.S. Government.

REFERENCES

Barstow, D.R., and Kant, E. (1974) Observations on the Interaction between Coding and

Efficiency Knowledge in the PSI Program Synthesis System, *Second International Conference on Software Engineering*, San Francisco.

Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R. (1972) *Structured Programming*. New York: Academic Press, Inc.

Floyd, Robert W. (1971) Toward Interactive Design of Correct Programs. *Computer Science report STAN-CS-235*. Stanford University.

Green, C. Cordell (1969) The Application of Theorem Proving to Question-Answering Systems. *Computer Science report STAN-CS-69-138 AIM-96, AD696394*, Stanford University.

Green, C.C. (1976) The Design of the PSI Program Synthesis System, *Second International Conference on Software Engineering*, San Francisco.

Green, C.C., and Barstow, D.R. (1975) Some Rules for the Automatic Synthesis of Programs. *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR.

Green, C.C., Waldinger, R.J., Barstow, D.R., Elschlager, R., Lenat, D.B., McCune, B.P., Shaw, D.E., and Steinberg, L.I. (1974) Progress Report on Program-Understanding Systems. *Computer Science report STAN-CS-74-444*, Stanford University.

Knuth, Donald E. (1973) Sorting and Searching. *The Art of Computer Programming* 3, Reading, Mass.: Addison-Wesley.

Knuth, Donald E. (1974) Structured Programming with GOTO Statements. *Computing Surveys* 6, 261-301.

Kowalski, R.A. (1974) Predicate logic as programming language. *Memo no. 70*, Department of Computational Logic, University of Edinburgh.

Martin, William A. (1971) Sorting. *Computing Surveys* 3, 147-174.

Ruth, Gregory R. (1974) Analysis of Algorithm Implementations. *Project MAC report MAC TR-130*. M.I.T.

Teitelman, Warren, *et al.* (1974) *INTERLISP Reference Manual*. Xerox Palo Alto Research Center and Bolt Berankek & Newman.

van Emden, M.H. (1974) First-order predicate logic as a high-level program language. *Report MIP-R-106*, Department of Machine Intelligence, University of Edinburgh.

Zahn, Charles T. (1974) A Control Statement for Natural Top-down Structured Programming, *Symposium on Programming Languages*, Paris.

DIALOGUE-TRANSFER OF KNOWLEDGE TO HUMANS

18

Representing Knowledge about Mathematics for Computer-Aided Teaching, Part I — Educational Applications of Conceptualizations from Artificial Intelligence*

Robert B. Davis

University of Illinois
Urbana-Champaign, Illinois

The first of these two articles deals with taking ideas from artificial intelligence and applying them in the analysis of the behavior of human students—that is to say, borrowing from A.I. in order to consider human learning and teaching. The second article looks specifically at some of the courseware being developed and tested for the PLATO computer system, with a focus on the ways in which PLATO authors try to achieve reasonably "human" behavior from a teaching computer system.

THE INFLUENCE OF BASIC CONCEPTUALIZATIONS

Scientific work impacts on practice in two ways: by the application of specific research results, and by the broader tendency to try to analyze reality in terms of general conceptualizations. The latter, though often "invisible" because of its ubiquity, is probably the more important of the two. (Consider, for example, how the treatment of alcoholics is affected by whether alcoholism is considered a crime or a disease; or how the treatment of neuroses and psychoses is influenced by whether these are conceived of as the action of the devil, or genetic, or physiological, or behavioral; or how early childhood educational programs are influenced by whether the main phenomena are considered as genetically determined (e.g., Jensen), or environmentally determined (e.g., Bereiter), or shaped by maturational processes (e.g., "Piagetians"). [Cf. (Polanyi, 1958; Koestler, 1959; Davis, 1967A)].

During the 1960's, the basic conceptualizations used by many teachers and curriculum designers were essentially Skinnerian. In fact, however, this fundamental conceptualization predates Skinner. In 1924 and 1925 Sidney Pressey demonstrated a teaching machine, for which (in 1927) he published a theoretical justification as follows:

*The research and development work reported here was financed by NSF grant No. PES-74-12567.

> It should be noticed, first of all, that the apparatus is so built as to function . . . in accordance with what is now known concerning the learning process . . . the "law of recency" operates to establish the correct answer in the mind of the learner, since always the last answer chosen (before the drum revolves to the next question) is the right answer. The correct response must almost inevitably be the most frequent, since the correct response is the only response by which the learner can go on to the next question; and since whenever a wrong response is made, it must be compensated for by a further correct reaction. The "law of exercise" is thus automatically made to function to establish the right response. Since the learner can progress only by making the right reaction, he is penalized every time he makes a wrong answer by being required to answer the question one more time, and is rewarded for two consecutive right responses by the elimination of that question, the "law of effect" is constantly operating to further the learning. Finally, certain fundamental requirements of efficiency in learning are met. The learner is instantly informed as to the correctness of each response he makes (does not have to wait until his paper is corrected by the teacher). His progress is made evident to him by the progressive elimination of items. And—most important of all—there is that individual and exact adjustment to difficulty mentioned at the beginning of the paper, by which wasteful overlearning is avoided and each item returned to until the learner has mastered it. [A recent re-issue of Pressey's 1927 article appears in (Lumsdaine and Glaser, 1960) cf. pp. 44-45].

The Skinner (or, more accurately, Pressey) conceptualization (which is also often called the "behaviorist" conceptualization, although this word has so many different meanings as to be misleading in many cases) has had a dominant effect on many teachers, probably because they studied it in "methods of teaching" or educational psychology courses during their own training. Human teachers inevitably deviate from the precise balance of Pressey's or Skinner's machines, but the basic conceptualization of learning is often quite recognizable, nonetheless. Consider, for example, the following description of teacher behavior, observed and recorded in the 1960's:

> in one high school of good reputation a teacher recently spent two consecutive 45 minute periods writing examples like this
>
> $$x^2 \cdot x^5 =$$
>
> on the blackboard and going around the room, letting each child in turn give an answer to a question of this type:
>
> $$x^2 \cdot x^5 = x^7$$
>
> $$p^2 \cdot p10 = p12$$
>
> $$x^3 \cdot x^2 = x^5$$
>
> $$x^2 \cdot x^6 = \_\_\_\_$$

> and so on. She construed that she was "teaching exponents"... (Davis, 1967B, p. 12)

Since the students (who, in fact, were top-track "gifted" students) were probably able to answer at least 3 such questions per minute, two 45-minute periods presumably allowed them to answer at least 270 questions of this type—seemingly a striking compliance with the "law of exercise", especially since the students got nearly every answer correct.

Of course, there has long been a strong "anti-behaviorist" view of education, [(cf., e.g., (Rising and Kaufman, 1973); also (Davis, 1959; Halmos, 1975; Moise, 1975; Piranian, 1975)], but these dissenting voices have been muted somewhat by the lack of any striking conceptualization of their own. It seems likely that artificial intelligence conceptualizations may now begin to fill this void.

What should have been a declaration of independence from Pressey's conceptualizations must, surely, be the Minsky and Papert monograph entitled *Artificial Intelligence Progress Report* (Papert and Minsky, 1972). Unfortunately, this monograph seems to be far less well-known than it deserves to be probably because it represents so extreme a challenge to the conventional wisdom that it is not well understood within the education community. Papert and Minsky argue that the Pressey conceptualization may, indeed, describe *something*, and in order to say more precisely what this "something" is, they invent a new category labelled *terminal learning*. Specifically, Minsky and Papert argue:

> There is a large literature concerned with clustering methods, scaling, factor analysis, and optimal decision theories, in which one finds proposals for programs that "learn" by successive modifications of numerical parameters. An outstanding example of this is seen in one of the well-known programs of A. Samuel, that plays a good game of Checkers. Other examples abound; all perceptron-like "adaptive" machines, all "hill-climbing" optimization programs, most "stochastic learning" models using reinforcement.
>
> Within the classes of concepts that these machines can represent, that is, describe as rather literal "sums" of already programmed "parts"—the learning abilities are effective and interesting. However, the descriptive powers of these quasi-linear learning schemes have such peculiar and crippling limitations that they can be used only in special ways. For example, we can construct, by special methods, a perceptron that could learn either to recognize squares, or to recognize circles. But the same machine would probably not be able to learn the class of "circles or squares"! It certainly could not describe (hence learn to recognize) a relational compound like "a circle inside a square".

These limitations are very confining. It is true that such methods can be useful in "decision-making" and diagnostic situations where things are understood so poorly that a "weighted decision" is better than nothing! But we think it might be useful to put this in perspective by assigning it as an example of a new concept of TERMINAL LEARNING. The basic problem with this kind of "learning program" is that once the program has been run, we end up only with numerical values of some parameters. The information in such an array of numbers is so homogeneous and unstructured–the "weight" of each "factor" depends so much on what other factors are also involved in the process–that each number itself has no separate meaning. We are convinced that the results of experience, to be useful to "higher level processes", must be summarized in forms that are convertible to structures that have at least some of the characteristics of computer programs–that is, something like fragments of programs or descriptions of ways to modify programs. Without such capabilities, the simple "adaptive" systems can "learn" some things, to be sure, but they cannot learn to learn better! They are confined to sharpening whatever "linear separation" or similar hypotheses they are initially set to evaluate. A terminal learning scheme can often be useful at the final stage of a performance or an application, but it is potentially crippling to use it within a system that may be expected later to develop further.

One could make similar criticisms of another aspect of the adaptive "branch and bound" procedures found in most game-playing and other heuristic programs that follow the "look-ahead and minimax" tradition. Suppose that in analyzing a chess position we discovered that the KB-2 square is vulnerable to a rook-queen fork by moving a knight to that square. The traditional program returns a low numerical value for that position. What it really should do is return a description of why the position is bad. Then the previous plausible-move generator can be given a constructive suggestion: look for moves that add a defense to that square, or threaten one of the attacking pieces, etc. Subsequent exploration will discover more such suggestions. Eventually, these conditions may come to conflict logically, e.g., by requiring a piece to attack two squares that cannot both lie in its range. At this point, a deductive program could see that it is necessary to think back to an earlier position. Similarly, a description of that situation, in turn, could be carried further back, so that eventually the move generator can come to work with a knowledgeable analysis of the strategic problem. Surely this is the sort of thing good players must do, but no programs yet do anything much like it.

This argument, if translated into technical specification, would say that if a chess program is to "really" analyze positions it must first have descriptive methods to modify or "update" its state of knowledge. Then it needs ways to "understand" this knowledge in the sense of being able to make inferences or deductions that help decide what experiments next to try. Here again, we encounter the problem of "common sense" knowledge since, although some of this structure will be specific to chess, much also belongs to more general principles of strategy and planning.

People working on these homogeneous "adaptive learning" schemas (either in heuristic programming or in psychology) are not unaware of this kind of problem. Unfortunately, most approaches to it take the form of attempting to generalize the coefficient-optimizing schema directly to multi-level structures of the same kind, such as n-layer perceptrons. In doing so, one immediately runs into mathematical problems: no one has found suitably attractive generalizations (for n levels) of the kinds of convergence theorems that, at the first level, make perceptrons (for example) seem so tempting. We are inclined to suspect that this difficulty is fundamental–that there simply do not exist algorithms for finding solutions in such spaces that operate by successive local approximations. Unfortunately we do not know how to prove anything about this or, for that matter, to formulate it in a respectably technical manner.

We could make similar remarks about most of the traditional "theories of learning" studied in Psychology courses. Almost all of these are involved with the equivalent of setting up connections with the equivalent of numerical coefficients between "nodes" all of the same general character. Some of these models have a limited capacity to form "chains of responses", or to cause some classes of events to acquire some control over the establishment of other kinds of connections. But none of these theories, from Pavlov on, seem to have adequate ability to build processes that can alter in interesting ways the manner in which other kinds of data are handled. These theories are therefore so inadequate, from a modern computation-theory view, that today we find it difficult to discuss them seriously. Why, then, have such theories been so persistently pursued? The followers were certainly not naive about these difficulties. One influence, we think, has been a pervasive misconception about the role of multiple trials, and of "practice", in learning. The supposition that repeated experiences are necessary for permanent learning certainly tempts one to look for "quantitative" models in which each experience has a small but cumulative effect on some quantity, say, "strength-of-connection".

In the so-called "stimulus-sampling" theories we do see an attempt to show how certain kinds of one trial learning processes could yield an external appearance of slow improvement. In this kind of theory a response can become connected with many different combinations of stimulus features or elements as a result of sampling processes. In each learning event a new combination can be tried and tested. This is certainly closer to the direction we are pointing. However, we are less interested in why it takes so many trials to train an animal to perform a simple sequence of acts and more interested in why a child can learn what a word means (in many instances) with only a single never-repeated explanation.

What is the basis for the multiple-trial belief? When a person is "memorizing" something he may repeat it over and over. When he practises a piece of music he plays it over and over. When we want him to learn to add we give him thousands of "exercises". When he learns tennis he hits thousands of balls.

Consider two extreme views of this. In the NUMERICAL theory he moves, in each trial, a little way toward the goal, strengthening the desired and weakening the undesired components of the behavior. In the SYMBOLIC view, in each trial there is a qualitative change in the structure of the activity—in its program. Many small changes are involved in debugging a new program, especially if one is not good at debugging! It is not a matter of strengthening components already weakly present so much as proposing and testing new ones.

The external appearance of slow improvement, in the SYMBOLIC view, is an illusion due to our lack of discernment. Even practicing scales, we would conjecture, involves distinct changes in one's strategies or plans for linking the many motor acts to already existing sequential process-schema in different ways, or altering the internal structures of those schemas. The improvement comes from definite, albeit many, moments of conscious or unconscious analysis, conjecture, and structural experiment. "Thoughtless" trials are essentially wasted.

To be sure, this is an extreme view. There are, no doubt, physiological aspects of motor and other learning which really do require some repetition and/or persistence for reliable performance. Our point is that the extent of this is really quite unknown and one should not make it the main focus of theory-making, because that path may never lead to insight into the important structural aspects of the problem. In motor-skill learning, for example, it is quite possible one needs much less practice than is popularly supposed. It

> takes a child perhaps fifteen minutes to learn to walk on stilts. But if you tell him to be sure to keep pulling them up, it takes only five minutes. Could we develop new linguistic skills so that we could explain the whole thing? We might conjecture that the "natural athlete" has no magical, global, coordination faculty but only (or should we say "only"!) has worked out for himself an unusually expressive abstract scheme for manipulating representations of physical activities [Minsky and Papert, 1972].

In the next section we use some of the very simplest A.I. ideas in order to analyze student performance data collected by Dr. Stanley Erlwanger.

CHILDREN'S ARITHMETICAL ERRORS

One of the largest studies of student errors ever undertaken was completed recently by Dr. Stanley Erlwanger, at the Curriculum Laboratory of the University of Illinois. Erlwanger used an interview procedure to observe the mathematical behavior of children in grades 4, 5, and 6. The interview data–a few examples of which we reproduce below–showed a very large proportion of wrong answers to simple arithmetical questions, and some bizarre misconceptions about arithmetical ideas. It is especially provocative to note that the children's teachers were largely unaware of these misconceptions, and in most cases believed that their students were making good progress in learning arithmetic.

We have been analyzing Erlwanger's data for the past year, and have yet to find a single exception to the rule that the children's errors consisted of incorrectly selecting, or incorrectly sequencing, small "mini" processes that were themselves correct. Moreover, there was very great consistency in the children's (wrong) methods.

Consider, for example, these excerpts from audiotaped interviews with a sixth grader named Benny (who had been identified by his teacher as doing well in mathematics; indeed Benny was thought by the teacher to be one of the most successful students in the class):

> Erlwanger: How would you write 2/10 as a decimal or decimal fraction?
> Benny: One point two (writes: 1.2)
> Erlwanger: And 5/10?
> Benny: (writes: 1.5)

[These interview transciptions were taken from (Davis, 1973).]

Benny's consistency is probably immediately clear; to emphasize the point, consider eight further conversions made by Benny:

$$\frac{400}{400} \rightarrow 8.00$$

$$\frac{9}{10} \rightarrow 1.9$$

$$\frac{429}{100} \rightarrow 5.29$$

$$\frac{3}{1000} \rightarrow 1.003$$

$$\frac{1}{8} \rightarrow .9$$

$$\frac{1}{9} \rightarrow 1.0$$

$$\frac{4}{6} \rightarrow 1.0$$

$$\frac{4}{11} \rightarrow 1.5$$

Consider the "mini" procedures that Benny uses: in the first example, for instance, he converted

$$\frac{2}{10}$$

to

$$1.2$$

Here are some of the "mini" procedures used:

Mini-procedure #1. The numeral 12 (used in constructing the answer, prior to inserting the decimal point) *is*, in fact, obtainable from the numerals 2 and 10 by a correct arithmetical operation

$$\begin{array}{r} 2 \\ +\,10 \\ \hline \end{array}$$

Does Benny use this same mini-procedure, in this same way, in the other examples? A quick check of all other examples shows that, indeed, he does; for instance, in converting

$$\frac{3}{1000}$$

to

$$1.003,$$

he uses the correct addition

$$\begin{array}{r} 3 \\ +\ 1000 \\ \hline 1003, \end{array}$$

and in writing

$$\frac{4}{11}$$

as

$$1.5$$

he uses

$$\begin{array}{r} 4 \\ +\ 11 \\ \hline 15 \end{array}$$

There are no exceptions.

Stimulus similarity: notice that the mini-procedure which should have been used ought to be a response to (say) the visual stimulus

$$\frac{4}{11},$$

whereas the mini-procedure that *was* used would have been a *correct* response to the visual stimulus

$$\begin{array}{r} 4 \\ +\ 11 \\ \hline \end{array}$$

(or even, in many books,

$$\begin{array}{r} 4 \\ \underline{11}\ ). \end{array}$$

The similarity of

$$\frac{4}{11}$$

to

$$\frac{4}{11}$$

is immediately apparent. This similarity of visual stimuli is a persistent characteristic throughout Erlwanger's data.

Mini-procedure #2 (inserting a decimal point). The final answers, in these examples, are obtained by a concatenation of two mini-procedures: obtaining the digits (mini-procedure #1), and inserting a decimal point (mini-procedure #2). While it is easy to obtain a complete (and credible) description of mini-procedure #1, from which there are no deviations, we have not been able to do quite so well for mini-procedure #2. To be sure, a fractional notation such as

$$\frac{2}{10}$$

will in most cases require a decimal point in the corresponding place-value numeral, and Benny *always* inserts one. In this respect he is again absolutely consistent. (With hindsight vision, one wishes that Erlwanger had posed a fraction such as

$$\frac{12}{4}$$

to Benny.)

We have not been able to identify any simple, credible rule which would determine *where* Benny places the decimal point. He *almost* uses the rule "place the decimal point to the right of the first digit." (Call this "mini-procedure 2-A.") There is only one exception to this rule, namely

$$\frac{1}{8} = .9$$

We *can* explain every case if we consider what the consequences of using mini-procedure 2-A would be in this case, namely

$$\frac{1}{8} = 9.$$

From looking at a large amount of Erlwanger's data [(Erlwanger, 1973), plus several further articles to appear in *The Journal of Children's Mathematical Behavior*; see also (Davis and Greenstein, 1969)], it seems clear that children make these "bizarre" errors only when they are manipulating symbols that are meaningless to them. Consider, for example, the following episode [Jennifer is described as "an alert 8-year old, nominally a third grader, but actually attending

a non-graded school"; her teacher *was* aware that Jennifer was having trouble with arithmetic, but believed the trouble to be localized to recent new work on division, whereas careful interviews showed the trouble to extend much further back into Jennifer's past history]:

Interviewer: shows Jennifer a paper with the problem

$8\overline{)4808}$ written on it.

Jennifer: 8 doesn't go into 4, so you have to say that 8 goes into 48 6 times.

[she writes]

$$\begin{array}{r} 6 \\ 8\overline{)4808} \end{array}$$

[Jennifer had centered "6" over the "48"; the interviewer suggested that, since the entire 48 had now been dealt with, it might be a safer procedure to write the 6 over the "8" in "48"; Jennifer made this change.]

Interviewer: All right. What do you do next?

Jennifer: The zero doesn't count for anything, so you say "8 goes into 8 once."

[she writes]

$$\begin{array}{r} 6\ 1 \\ 8\overline{)4808} \end{array}$$

Interviewer: [writes, *without pronouncing* them,

$8\overline{)4808}$ $8\overline{)488}$.]

[says] Would these two problems be the same?

Jennifer: Yes

Interviewer: How about this example?

[writes] $8\overline{)800}$

Jennifer: Eight goes into eight once.

[she writes]

$$\begin{array}{r} 1 \\ 8\overline{)800} \end{array}$$

Interviewer: Which of these is the largest?

[writes] 8 80 800

Jennifer: They're all the same. Zero doesn't make any difference.

Interviewer: How old are you?

Jennifer: Eight

Interviewer: Do you have any older brothers or sisters?

Jennifer: A brother... [whose age was revealed, by further discussion, to be 9] and a sister... [whose age was revealed to be 11. (The interviewer was seeking some numbers, meaningful to Jennifer, that involved digits of zero.)] ...

Interviewer: Can you write "ten"?

Jennifer: [writes] 10.

Interviewer: Which is larger?

[writes] 1 10

Jennifer: They're both the same, because zero doesn't make any difference. [Suddenly a *real* expression of comprehension lit up her face:] Oh! That's *one* and that's *ten*!

The description of the interview continues:

At this point Jennifer responded in a fashion very common among children, as David Page has pointed out very accurately: with no prompting from the interviewer, Jennifer eagerly and immediately turned her attention back to the *preceding* problem—we might call this the "solecism effect"—and, realizing that she must have answered incorrectly there, she now straightened out the difference between the written symbols

8 80 800.

(Davis and Greenstein, 1969).

Modified second mini-procedure. The Jennifer example (which is typical), shows that children will use bizarre rules in dealing with meaningless symbols, but, *the moment the symbols become meaningful to them*, they drop these bizarre rules, even to the extent of going back and revising recent performances (as Jennifer went back and revised her answer about 8 vs. 80 vs. 800). Hence, we *could* explain Benny's single exception—writing

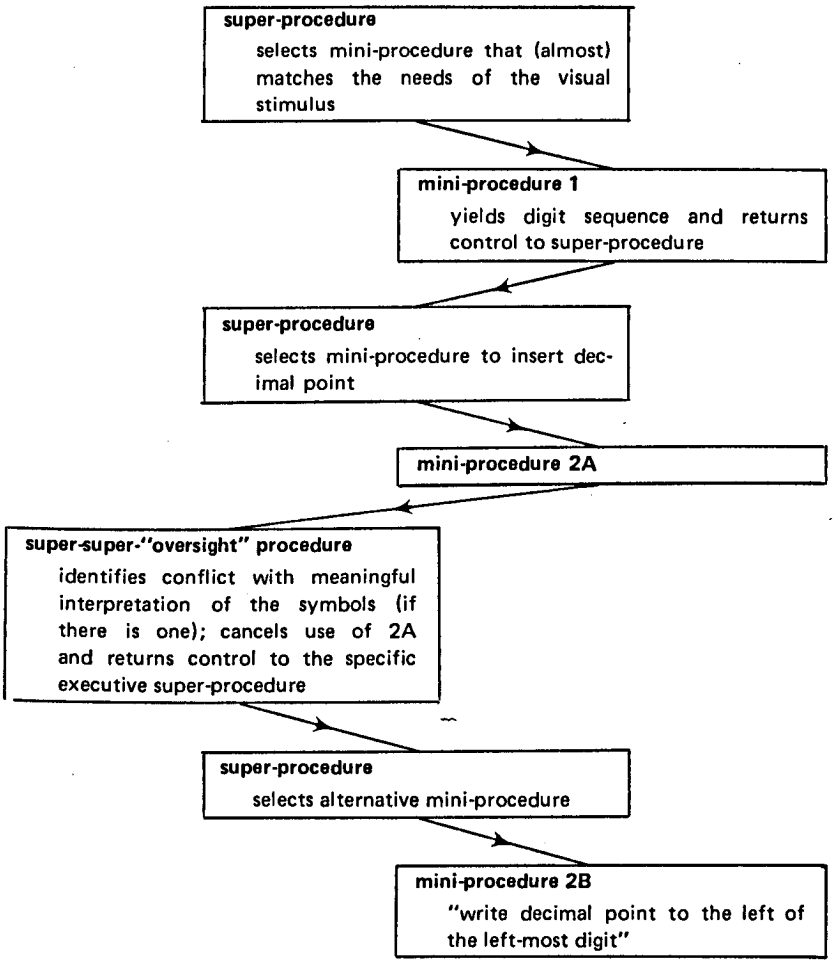
$$\frac{1}{8} = .9$$

where a consistent use of rule 2-A would have yielded

$$\frac{1}{8} = 9.$$

—as an instance of this "don't do it if it conflicts with a meaningful interpretation of the symbols" procedure.

One might attempt to diagram Benny's entire procedure, in response to any one of these "fractions-to-decimal" conversion problems, as:


super-procedure
selects mini-procedure that (almost) matches the needs of the visual stimulus
mini-procedure 1
yields digit sequence and returns control to super-procedure
super-procedure
selects mini-procedure to insert decimal point
mini-procedure 2A
super-super-"oversight" procedure
identifies conflict with meaningful interpretation of the symbols (if there is one); cancels use of 2A and returns control to the specific executive super-procedure
super-procedure
selects alternative mini-procedure
mini-procedure 2B
"write decimal point to the left of the left-most digit"

[We have not attempted to expand the diagram to achieve a complete "flow-chart" type of diagram, although how this could easily be done should be apparent.]

Whether this modified rule meets the requirement of credibility is left to the reader; unfortunately, this interpretation was just made in the spring of 1975, too late to pose further questions to Benny to see if all exceptions had now been eliminated. (Our guess is that they have not been; it seems likely that Benny has a large repertoire of "emergency alternative procedures" that can be invoked when his normal procedures yield a conflict with meaning, or (as we shall see below) with other external data.)

Do Benny's Verbal Explanations Help Us to Understand How He Selects Mini-Procedures? Erlwanger's interviews contain two different kinds of data: on the one hand, the explicit mathematical responses of the children, but on the other hand, also the children's remarks, discussions, and explanations of what they were thinking, why they selected specific answers, and so on. Do these explanation marks help us to understand how the children select mini-procedures?

Someday they may, if anyone ever figures out how to interpret them, but for the moment we are unable to make very much use of them.

Here are some typical excerpts from Erlwanger's notes on Benny's explanatory remarks:

> Benny: [asked to explain how he did the
>
> $$\frac{5}{10} = 1.5$$
>
> conversion:]
>
> The one stands for ten; [pointing, and implying sequential order] the decimal; then there's five... [pause] ... shows how many ones. (Erlwanger, 1973)

As an *explanation* of how Benny was thinking, this has not thusfar proved helpful. The picture changes, however, *if we consider Benny's verbal production as a further example of a product created by his intelligence procedures.* The task Benny was confronted with was to produce an oral sequence in response to the interviewer's query. Once again we see him using mini-procedures which are themselves correct, but wrongly applied in the present situation. Thus

"the one stands for ten"

would be a correct statement if applied, say, to

2,913 ;

it is wrong only in that it was wrongly selected for application to

1.5

The identification of the decimal point, though unenlightening as to *why* it was put where it was, was nonetheless an unequivocally correct oral response to the visual stimulus he was looking at at the time: it was, indeed, a decimal point. He said he was looking at a decimal point, and he was.

Finally, the remark that

"there's five ... shows how many ones"

would be a correct response to the visual stimulus

15

but is wrongly selected as a description of

1.5.

Hence, if the *content* of Benny's explanatory remarks has not thusfar proved helpful, their *form* provides us with additional instances of output from Benny's mental processes–and, seen in this way, they provide further confirmation of our general rule, namely:

> Outputs, whether written or oral, are produced by super-procedures that select a sequence of mini-procedures, apparently under a requirement of non-conflict with "meanings" whenever meanings are present. The individual mini-procedures are *always* correct ones. The errors come from the super-procedure incorrectly selecting which mini-procedures to use. Incorrectly selected mini-procedures are related to appropriate mini-procedures by a similarity of the visual stimuli that should elicit them.

Perhaps, however, the relationship of the oral productions ("explanations") to the written productions ("answers") should not be dismissed without further comment. Returning to the interview with Jennifer, we see at once the special role being played by her remarks:

> "The zero doesn't count for anything. . ."
> "Zero doesn't make any difference."

Considered as *ideas*, these ideas would tend to justify her choice of mini-procedures; hence one is tempted to say that these "ideas" are influencing that choice. But one who wishes to interpret all of this behavior in terms of super-procedures selecting mini-procedures can easily do so: given the visual stimulus

$$8/\overline{4808}^{\,6},$$

Jennifer is confronted with the "0", and she selects a mini-procedure to deal with it–namely, the verbal rule "zero doesn't count for anything."

This leaves unanswered the question of how this verbal rule then influences the selection of the mini-procedure that governs the subsequent writing; evidently, if we seek maximum homogeneity in our theory, we might assign to this *oral* statement essentially the same role that we assign to the visual stimulus: given this visual stimulus *and* this oral stimulus (since, presumably, Jennifer "heard" what she "said"), what mini-procedure should be selected?

This begins to verge upon an interpretation of *implication* in human thought as an S-R phenomenon of producing inferences by responding to premises as stimuli. We do not at present have any satisfactory theory to discuss this part of the thinking process.

However, to underline what we *do* have, notice once again that these state-

ments, regarded as *productions*, still conform to our rule. "Zero doesn't count for anything," and "zero doesn't make any difference" are both *correct* statements (though dangerously misleading ones), *provided* they are applied to appropriate situations, such as

$$7 + 0 = 7$$
$$x + 0 = x$$
$$1.0000$$
$$\emptyset \qquad \text{(the empty set)},$$

and so on. The aren't *wrong* mini-productions, they are *wrongly selected* for this context.

ACCOMMODATION

Piagetian psychology uses the word "accommodation" to name the phenomenon of modifying one's ideas in order to achieve a better match with external reality (or, indeed, in order to eliminate an internal contradiction in one's own thinking, if one suddenly becomes aware of one). Accommodation presumably occurs mainly in response to feedback data that raises serious questions about one's thinking.

A precise definition of the phenomenon of "accommodation" cannot be given, at present, because Piagetian psychology, having not enjoyed the advantages of A.I. conceptualizations, has not described the internal structure of mental data processing in terms of "procedures" and other similar ideas.

Presumably, when Jennifer had the sudden realization that she had been *wrong* to think that 1 and 10 meant the same thing–a realization directly observable as a "flash of insight" or "an expression of revelation that an observer could hardly fail to notice" -- she was experiencing "accommodation." Presumably, too, this refers not to the change in her *behavior*–e.g., going back to revise her remarks about 8 and 80 and 800–but rather to some change that she *may* have made in her mental information-processing procedures. Of course, the evidence does not prove that she made any such change at all; the change in her output may have come entirely from a change in the *input*–she had just taken in the discrepant fact that

1

is pronounced "one", whereas

10

is pronounced "ten".

To be sure that Jennifer's *procedures* had been changed we would need to have data from a subsequent time when she was asked a similar sequence of questions, and such data is not available. However, any experienced teacher will probably guess that Jennifer was deeply impressed by the trap into which her

thinking led her; on a subsequent occasion she will attend more to the way you *pronounce* "8" or "80" or "800". (Notice that such a change is easily explained as an adjustment of parameters in the sense of Minsky and Papert's "terminal learning," although this may not be the best way to conceptualize it.)

That something like "accommodation" is important to these children is clear from the interview data. Here are some examples:

> . . .[Benny] was fully aware of the fact that. . . [his method of converting a fraction to a decimal] will give identical decimals arising from many different fractions, but he did not appear to think that there was anything wrong with that, as illustrated by this excerpt:
>
> Erlwanger: And $\frac{4}{11}$?
>
> Benny: 1.5
>
> Erlwanger: Now, does it matter if we change this $\left[\frac{4}{11}\right]$ and write instead [writes]
>
> $$\frac{11}{4}?$$
>
> Benny: It won't change at all; it will be the same thing. . .1.5.
>
> Erlwanger: How is that possible? Four elevenths is the same as eleven fourths?
>
> Benny: Ya. . . because there's a ten at the top. So you have to drop that 10. . . take away the 10; put it down at the bottom
>
> [writes:
>
> $$\frac{11}{4}$$
>
> $$\frac{1}{14}\,]$$
>
> Then there will be a *one* and a *four*. So really it will be [points to
>
> $$\frac{1}{14}\,].$$
>
> So you have to add these numbers up, which will be 5; then ten; so, 1.5.

Benny's thinking has a remarkable degree of internal consistency, as many

examples reveal. Presumably this is the result of repeated instances of accommodation. For example, asked

Erlwanger: How would you write .5 as an ordinary fraction?

Benny replies

Benny: .5 . . . it will be like this . . .

[writes $\frac{3}{2}$]

or else

[writes $\frac{2}{3}$]

or anything, as long as it comes out with the answer 5, because you're adding them. (Erlwanger, 1973, p. 9)

How is it possible that, despite enough accommodation to produce great internal consistency, Benny has not achieved a better match with external reality?

The answer seems to lie in the kind of feedback that Benny received from the outside world. In grades 2 through 6, Benny had been a student in a so-called "individualized" instructional program in arithmetic. Each student in the class worked alone, receiving a pamphlet that introduces a new task by showing a sample problem, such as:

Write the correct decimal numeral for each mixed fraction.

$$6\frac{24}{100} = \underline{6.24}$$

$$9\frac{35}{1000} = \underline{9.035}$$

$$27\frac{15}{1000} = \underline{27}$$

(Erlwanger, 1973, p. 20)

The student tried to figure out what was wanted, and to write it down. He then took his pamphlet to a para-professional teacher's aide, who checked it against an answer key (hence these teacher's aides were called "checkers"), and returned it to the student with each answer marked wrong if it failed to match the answer key. The cycle then repeated, with the same topic if many answers were marked wrong, and with the next pamphlet otherwise.

Thus, too little "explanatory" information was being brought back to the student. *It is, amazingly, a precise external realization of what Minsky and Papert have labelled "terminal learning."*

Benny had, somehow, to reconcile himself to this whole cycle, with its unexplained "rights" or "wrongs". He did this in two parts. First, he explained the *situation* (one might almost say, he explained the epistemology) as follows:

> Benny: They [the checkers] mark it wrong because they just go by the key. They don't go by if the answer is true or not. They go by the key. It's like if I had $\frac{2}{4}$; they wanted to know what it was, and I wrote down one whole number, and the key said a whole number, it would be right–no matter that it was wrong. (Erlwanger, 1973, p. 16)

But Benny still needs to explain just how some answers are "really correct," despite external evidence against them. He does so by one of the most audacious theories imaginable!

> Erlwanger: It [i.e., the process of finding answers] seems to me like a game.
>
> Benny: [Emotionally] Yes! It's like a wild goose chase!
>
> Erlwanger: So, you're chasing after the answers that the teacher wants?
>
> Benny: Ya! Ya!
>
> Erlwanger: Which answers would you *like* to put down?
>
> Benny: [Shouting] Any! As long as I knew it could be the right answer. You see, I am used to checking my own work; and I am used to the key. So I just put down $\frac{1}{2}$ because I don't want to get it wrong.
>
> Erlwanger: Mmm. . .
>
> Benny: Because if I put $\frac{1}{4}+\frac{1}{4}$, they'll mark it wrong. But it would be right! You agree with me there, O.K.? If I put $\frac{2}{4}$, you agree there. If I put $\frac{1}{2}$, you agree there, too. They're all right! (Erlwanger, 1973, p.16)

In other words, Benny notices that there may be many expressions that look different, all of which are nonetheless correct. Of course, the checkers will mark most of these wrong. He generalizes this idea, and believes that *all* of his answers are correct, although they don't *look* like the answers in the key. And, indeed, as we have seen, Benny has worked out transformation rules that allow him to

write his "wrong" answers in a new form, as "right" answers. He believes that all of this is the same phenomenon as rewriting

$$\frac{1}{4}+\frac{1}{4}$$

or

$$\frac{2}{4}$$

and then re-writing this as

$$\frac{1}{2} \ .$$

Somewhat similar rationalizations had been created by other children. Consider the case of Mat. Asked to multiply 200 by 100, Mat said he could do it several different ways, and wrote

200	200	200
x 100	x 100	x100
20000	2000	200

Erlwanger: Now, which is the right answer?

Mat: [Shouting] *They're all right! It's just that the numbers are in different positions!*

Erlwanger: Would it work this way in addition and subtraction... and so forth?

Mat: [pause] Well, let's take 200 plus 100. [Mat wrote $\begin{array}{r} 200 \\ +\ 100 \end{array}$, $\begin{array}{r} 200 \\ +\ 100 \end{array}$, and $\begin{array}{r} 200 \\ +100 \end{array}$,]. *You wouldn't get the same answers now...* but...300...O.K? ...and 200...100... there... O.K. now [Mat wrote down his answers as 300, 2100, and 20100 from right to left, respectively].

Erlwanger: And you say they are all different?

Mat: Uh uh.

Erlwanger: Would they [booklets] use all these three methods in mathematics?

Mat: *Not at the same time...* maybe on *different pages...* ya.

Erlwanger: Now where would you come across an example like 200 + 100... like this... with the 1 below the first zero [i.e., 200 / + 100].

Mat: In addition.

Erlwanger: Have you ever done some like that?

Mat: I think so... ya, ya... I have. It's been in the third grade though... because I was in E addition in the third garde.

Erlwanger: Uh uh. Now suppose you... you had to take a test. O.K.? And they had this example down ...520 + 360 ... written across [i.e., horizontally]. How would you do it? [Mat worked as follows:

$$\begin{array}{r} 520 \\ +360 \\ \hline 880 \end{array}$$

Erlwanger: How would you know they wanted you to put it that way?

Mat: Well... another way to do it is to move this [520] down here [Mat wrote $\begin{array}{r} 360 \\ +520 \end{array}$]

Erlwanger: Why would you use that one?

Mat: *Because that's the way... that's the way it's usually written.*

Erlwanger: Pardon?

Mat: [somewhat excited] *That's the way they are!* You see, *in the booklet...* they... they... on the *first page sometimes they show you... like this goes here, this goes here, and this goes here* [pointing at each example].

Erlwanger: Oh... you mean they... they tell...

Mat: [interrupting] *They show you at the first [page] of the book. They don't tell you in the test...* but when you get it in the book, they *tell you how to do it.*

In all of this, we see some form of accommodation being used to reduce cognitive dissonance, though not always to achieve correct mathematical results. This appears to be a true use of inadequate feedback in the sense of Papert and Minsky's "terminal learning."

EXPLANATION OF A MYSTERY?

During the "new mathematics" curriculum innovations of the 1960's, one of the biggest successes was the remarkably effective teaching of David Page. Indeed, it was mainly Page's work that led Bruner to make his famous remark that

> . . .any subject can be taught effectively in some intellectually honest form to any child at any stage of development. (Bruner, 1962)

Page's successes (which, to some extent, were replicated by William Johntz in Berkeley, California, and by others) should have given rise to much discussion, analysis, and, if possible, widespread implementation. Instead, they tended to be dismissed with the remark that Page was such a great teacher that he could teach anything to anybody—with the usual implication that such a phenomenon could hardly be of any interest. The explanation of Page's success was hardly sought at all.

The present A.I.-type analysis of student errors suggests what may be the beginning of such an explanation.

While there were a number of recognizable teaching strategies that seemed to be important ingredients in Page's approach, one very important one was Page's response to student errors. If, for example, a student's response involved the error

$$4 \times 4 = 8,$$

Page would say: "Yes, four *plus* four would be eight. . .", or would ask: "How much is four *plus* four?" As Page pointed out, it nearly always occurs with elementary school children that, thus being asked "How much is four plus four?", they do *not* answer this most recent question, but immediately correct their wrong answer to the *preceding* question, e.g., by exclaiming: "It should be sixteen!"

Now, the Erlwanger data shows that schools seem to be having excellent success in developing correct mini-procedures. Student errors stem from malfunctioning of the super-procedures that must select and sequence the mini-procedures. (Indeed, observers have been arguing for years that schools have been teaching

$$x^2 \cdot x^5 = x^7$$

with such excessive repetition, with so little variation in the type of problem, and with such a total absence of heuristic analysis of problem-attack strategies, that the students arrive at college and write

$$x^2 + x^5 = x^7 .)$$ (Davis, 1967B, pp. 12-13)

Page, by contrast, was focussing precisely on the failures of the super-procedures. Indeed, one of his explanations of how to teach was: "A child never gives a wrong answer. He always gives a right answer to *some* question. The teacher's task is to figure out what question the child was answering, and call the child's attention to the distinction between that question and the one he was *supposed* to be answering." Erlwanger's data strongly suggests the need for such an emphasis.

SUMMARY

The time has come to employ A.I. conceptualizations in the analysis of human teaching and learning phenomena. One or two such possibilities have been illustrated, particularly the analysis of student errors in terms of correct mini-procedures incorrectly selected by a malfunctioning super-procedure, and the recognition that David Page's extraordinarily effective teaching did in fact focus on debugging these super-procedures, a matter ignored in much standard educational practice even today.

BIBLIOGRAPHY

Bruner, Jerome S. (1962) *The Process of Education*. Harvard University Press, Cambridge, Massachusetts.

Davis, Robert B. (1967) *Mathematical Teaching–With Special Reference to Epistemological Problems*, Monograph No. 1 of the Journal of Research and Development in Education, College of Education, University of Georgia, Athens, Georgia 30601.

Davis, Robert B. (1967B) *The Changing Curriculum: Mathematics*, Association for Supervision and Curriculum Development, NEA.

Davis, Robert B. (1959) *Solving Problems and Constructing Systems–Quadratic Equations and Vectors*. Report on an Orientation Conference for SMSG Experimental Centers, Chicago, Illinois, pp. 97-101.

Davis, Robert B. (1973) "Two Special Aspects of Math Labs and Individualization: Papért's Projects and Piagetian Interviews," Chapter Two in: Jon L. Higgins and Richard Lesh, Eds., *Cognitive Psychology and the Mathematics Laboratory*, Ohio State University, ERIC Center.

Davis, Robert B. and Greenstein, Rhonda (1969) Jennifer. *New York State Mathematics Teachers Journal*, XIX, 3, 94-103.

Erlwanger, Stanley H. (1973) Benny's Conception of Rules and Answers in IPI Mathematics. *The Journal of Children's Mathematical Behavior*, I, 2.

Erlwanger, Stanley H. (1974) *Case Studies of Children's Conceptions of Mathematics*, Ph.D. Thesis, University of Illinois.

Halmos, P.R. (1975) The Teaching of Problem-Solving. *American Mathematical Monthly*, 82, 5, 466-470.

Koestler, Arthur (1959) *The Sleepwalkers*, Macmillan, [re-issued in paperback by Grosset and Dunlap, 1963].

Lumsdaine, A.A., and Glaser, R., Eds. (1960) *Teaching Machines and Programmed Learning: A Source Book*. Washington, D.C., National Education Association of the United States.

Moise, E.E. (1975) The Problem of Learning to Teach. *American Mathematical Monthly*, 82, 5, 470-473.

Papert, Seymour, and Minsky, Marvin (1972) *Artificial Intelligence Progress Report:*

Research at the Laboratory in Vision, Language, and Other Problems of Intelligence. Artificial Intelligence Memo No. 252, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts.

Piranian, George (1975) The Promotion of Participation. *American Mathematical Monthly*, 82, 5, 474-476.

Polanyi, Michael (1958) *Personal Knowledge*, University of Chicago Press.

Rising, Gerald, and Kaufmann, Burt, Eds. (1973) special issue of *Educational Technology*, XIII, 11.

19

Representing Knowledge about Mathematics for Computer-Aided Teaching, Part II — The Diversity of Roles that a Computer Can Play in Assisting Learning

Robert B. Davis, Sharon Dugdale, David Kibbey, and Charles Weaver

Computer-Based Education Research Laboratory
University of Illinois at Urbana-Champaign

In Part I, we used a conceptualization from artificial intelligence to analyze human behavior in learning and teaching mathematics. In the present paper we shall be dealing with the PLATO computer system at the University of Illinois—probably the world's largest educational computer system—and specifically with the task of getting the most "human-like" performance possible from a computer-aided instruction (CAI) system. The methods employed thusfar at PLATO have used either some very simple A.I. programming, or have by-passed A.I. approaches and somehow returned the task to human beings, most often to the students themselves. Although no advanced A.I. has thusfar played much of a role in PLATO courseware, the potential role for it is very great.

In a recent doctoral thesis at the University of Illinois, to which we shall refer more extensively later, Resnick (Resnick, 1975) remarks that the vast majority of CAI courseware fits into one of three categories:

1. Courseware that attempts to break the subject-matter content down into "bite-sized" pieces and to sequence these, with no serious modeling of students beyond very simple measures of error-rate or the like;
2. Courseware that, in addition, attempts to make decisions on the basis of a few further aspects of individual student characteristics—for example, attempts to get at, and make use of, variations in cognitive style;
3. Courseware that attempts to recognize the student's individual internal cognitive structures—his understandings and misunderstandings—and to interact with these so as to promote their beneficial growth.

*The research and development work on PLATO described in this paper has been supported by National Science Foundation Contract No. USNSF C-723.

Resnick's focus is on students, rather than on A.I. methodology, and her category system clearly reflects this. Consequently, it isn't clear where, in her system, one would classify any elaborate *simulation* of, say, some physical or biological phenomenon, intended to give students "realistic" experience. Presumably it mainly falls under category 1.

The meaning of category 3 may deserve some discussion. The fundamental idea comes primarily from Piagetian psychology (Davis, 1972)–the idea that each of us builds up, in his own mind, his own models of reality. Things newly learned are fitted into this structure, where possible, with minimal modification of the structure itself (which the Piagetians call "assimilation"); where this is *not* possible, the new knowledge is ignored or severely distorted so as to make it fit (this would include Freudian *denial*), or else the person's internal cognitive structure is altered so as to make possible assimilation into the *new* cognitive structure (and this changing of the basic cognitive structure the Piagetians call *accommodation*). Examples abound in daily life and in the history of every intellectual discipline (Polanyi, 1958; Koestler, 1959; Kuhn, 1962).

Piagetian theory has been developing ever since 1921 (Flavell, 1963); hence it pre-dates artificial intelligence, and has in any case intended to focus on epistemology rather than on learning. It is not easy, and probably not possible, to give precise A.I. translations of Piagetian ideas; and when we look at examples of CAI courseware, classification becomes extremely imprecise, to say the least.

Nonetheless, the central idea is to attempt to identify key aspects of the learner's conceptual framework, and to use this identification in helping him to learn the new material.

The child's boner "The equator is an imaginary lion running around the earth" reveals a zoological assimilation of what had been intended as a geographical idea. When David Page says that the child never gives us a wrong answer, but rather the correct answer to some other question, and that the teacher's task is thus to identify this other question, and help the child to see how his question differs from the teacher's question, Page is advocating a teaching strategy that falls into Resnick's third category.

To Resnick's three categories of CAI, we would like to add a fourth. Courseware authors have often acted as if "mathematics" (or some other subject of interest) is a very simple *kind* of knowledge, easily enumerated in some fashion; moreover, they have created courseware as if this "content" is the direct and total business of learning.

Our own interest, at the moment, is the use of computers to teach mathematics to children in grades 4 through 9. In many senses this is an *ecological* problem. We have inserted about 50 PLATO terminals (see Figures 1-3) into the ecology of a dozen or so local classrooms; while one obvious and important question is what math, in the narrowest sense, are the children learning. There is another question that we consider very important: how is PLATO fitting into the ecology of these classrooms? Which of the things that normally go on in a classroom are being enhanced? Which of the things that normally go on in a classroom are being inhibited? In each case, is the change for the better, or not?

FIG. 1. A PLATO IV terminal in an elementary school classroom.

FIG. 2. What happens to the ecology of an elementary school classroom when PLATO terminals are added?

FIG. 3. Students involve themselves with PLATO in a variety of ways. . .

In a typical classroom, what *does* go on, anyhow? The list is vast, and has received surprisingly little comprehensive study. Some examples: a child does a piece of work, and proudly shows it to another child. Or he shows it to an adult. The adult may express praise. Another child, seeing the work, may feel discouraged that he can't do equally well. Or he may become fiercely determined to do even better. A child's work may turn out to contain important flaws. He may be embarrassed by this, and subsequently avoid that subject. Or, he may become determined to overcome the flaws. One child may help another. Two children may play a game. One child may observe another's work, and thereby get an idea that he subsequently develops himself. One child may argue with another. The other may look for flaws in the argument of the first.

The ecology of contemporary classrooms is complex and little understood (Davis, 1974). Are we selecting the most valuable parts to enhance by the presence of our computer terminals? Are the aspects that we inhibit unimportant to good learning?

Our fourth category of CAI courseware, therefore, is courseware that attempts to be "ecologically sound" and "ecologically diverse." It reaches for a variety of roles, and attempts to play them in the most "human" way possible—that is, it reaches for a performance that would be called good teacher behavior if it were done by a person instead of a machine. We shall presently delineate this more precisely.

PLATO

From the point of view of hardware and system software, the computer system with which we are working, PLATO IV, is an interactive time-sharing computer-assisted instruction system at the University of Illinois, Urbana. It consists of approximately 900 graphics terminals connected by phone lines and microwave links to a central computer facility at the Computer-Based Education Research Laboratory (CERL).

The system has the following properties which have done much to shape the instructional materials being developed: ample computing power and memory, good graphics capabilities, fast response times, simple terminal input devices, and an easy to use programming language with extensive and flexible answer judging, branching, and graphics capabilities.

For those interested, a more complete description of the system follows:

The computer

The central PLATO IV computer revolves around a Control Data Cyber 73-2 system.

Programs are executed in a time sharing mode in approximately 12 millisecond time slices, at a typical rate of 3-6 thousand machine instructions per second.

The central memory adds 65K 60 bit words of which programs may use about 10K words.

Between time slices, programs are stored in 2 million words of extended core storage.

Response times are typically under .2 seconds.

Long term storage of programs is on magnetic discs. Space on these discs is subdivided into so-called "lesson spaces."

The terminal

The main feature of the terminal is a transparent flat glass plasma panel 8½ inches square, with 512 x 512 individually addressable "dots" that can be turned on or off (each dot is, in fact, a tiny bubble of neon gas).

There are also 256 immediately addressable 8 x 16 dot characters of which 128 are hard wired, and 128 are programmable.

Line drawing functions are hard wired into the terminal.

Up to 2048 characters may be displayed on the screen at a rate of 180 characters per second.

Input devices

The usual input device is a keyset which consists of a standard typewriter keyboard (including upper case letters) plus a number of function keys.

Touch panel

In addition, a touch panel based on light sensitive diodes is available. The touch panel detects touches on the screen at any point of a 16 x 16 grid. Its value is the way in which it allows inputs by direct interaction with the screen.

Additional features

The terminal has other features which are not used for the elementary mathematics courseware:

1. A slide projector under control of the computer which projects directly on the plasma panel and can store up to 256 images in a single microfiche.
2. A random-access audio device which can store over 4000 separate messages of up to 21 minutes in length, each being a high-fidelity recording of a human voice, a musical instrument, heartbeat sounds as heard through a stethoscope, etc.

The TUTOR language

PLATO programs are written in a specially designed programming language, TUTOR, which was developed by Paul Tenczar and others at CERL for use in CAI.

The TUTOR language has good graphics facilities, including easy line and circle drawing, ability to rotate, move, and change the size of characters. Hard wired characters may also be rotated and changed in size. Service routines are available for the design of characters and for designing entire screen displays by using a moving cursor. The elaborate displays in many lessons are reflections of

TUTOR's graphics capabilities.

Perhaps the main feature of TUTOR is the flexibility and ease of judging student responses. Built-in features allow recognition of spelling errors, changes of order, synonyms, alternate phrases. Many alternative responses can be listed, judged correct or incorrect and the program can be branched accordingly. Information such as the number of wrong responses, which correct response was given, whether answers were judged incorrect because of spelling, or because of wrong word order, and so forth are kept automatically and in many cases reported to the student. Such information is useful to the programmers, and also provides feedback that helps prevent a student from gradually permuting a basically correct response (for example as incorrect spelling) to one totally incorrect.

In addition to the natural-language judging mentioned above, student algebraic responses can be dealt with especially easily. The system automatically reports a large number of different syntax errors in any algebraic expressions that form part of student responses.

A few further details of some of these features will be presented below.

TUTOR is useful for a diversity of purposes. It is simple enough that it can be used by elementary school students and by inexperienced teachers, yet it provides enough power to allow sophisticated programming that takes advantage of the large CDC machine to which all of the terminals are linked.

A FEW EXAMPLES

To give an indication of the level of A.I. programming that is incorporated in PLATO courseware (as we shall see, "courseware" is being used rather broadly), we mention a few examples, none of which are especially unusual. Recall that PLATO has an atypically great computing power for a CAI installation, and terminals with excellent graphic capabilities; this obviously influences courseware design.

Paul Tenczar's genetics lessons

A few years ago, biology students at the University of Illinois studied genetics in part via lab experiments with fruit flies. These experiments were always messy and time-consuming, and they gradually became prohibitively expensive; hence they were replaced by PLATO simulations. The main idea is obvious; symbolic representations for genes are stored in memory; they combine under random-number processes that match known biological probabilities, with allowance for dominant and recessive genes, sex-linked characteristics, lethal characteristics, etc. From this symbolic combination PLATO generates pictorial sketches of fruit flies with normal wings, with degenerate wings, black or pink eyes, etc. Not unusual as A.I., but somewhat ambitious by usual CAI standards. In the experiments students choose which flies to mate with which, as they would with real flies. (Bitzer, Sherwood, and Tenczar, 1973).

Carol Bennett's lunar landing module

Students can control retrorocket fire to "land" a depicted lunar landing vehicle on the depicted surface of the moon. Landing module behavior is governed by the usual equations of dynamics.

Richard Blomme's checker player

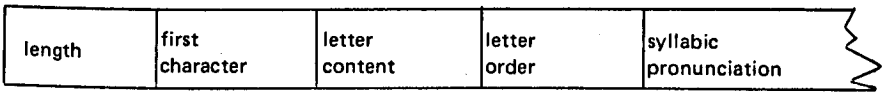
PLATO plays a respectable, but not powerful, game of checkers, programmed by Richard Blomme. Perhaps the most important point for practical CAI is that Blomme has achieved a level of play that makes PLATO an interesting but not invincible opponent, so that many humans enjoy playing against the machine time and time again.

Misspelling detector

Somewhat more unusual is the misspelled word detector created by Paul Tenczar and William Golden. In order to interpret student inputs appropriately, it is important for PLATO to distinguish a correct but misspelled word from a word which has some other meaning. The basic method is an adaptation of a biological method for comparing strings of genes. Comparing the n^{th} letters A_n and B_n of a word and a possibly misspelled version of that same word would clearly fail, since for example, if word A is: misspelled and word B is: mispelled, then $A_n \neq B_n$ for $3 < n \leqslant 10$, so that, by such a test, the words seem very dissimilar. What is done is to replace both words A and B by two theoretical "words" A′ and B′, where each "letter" in the theoretical words denotes some important aspect of the original word. It then becomes possible to compare A′ and B′ by comparing A_n' with B_n' for each value of n.

The method by which the theoretical words A′ and B′ are constructed deserves mention. Tenczar and Golden write:

> Almost any mapping scheme will work well if it uses *several* human criteria to set bits. . . .no one criterion appears sufficient to do the mysterious thing which humans do when they recognize words. Rather, we should use as many features of words as we can think of. . . We must ask ourselves, what do we see when we look at words? Here is our current vision:



> Each of these fields is assigned a bit length determined by our subjective feeling of the importance of the field (Tenczar and Golden, 1972, p. 6).

The method of setting bits in the "length" field is described by Tenczar and Golden as follows:

> The length of a word creates an obvious impression. A two letter

word cannot be a misspelling of an eight letter word! As a first consideration, one might attempt to use the binary representation of the word's length in this field.

The following table shows that that is undesirable:

Word length	Binary Representation	Better Representation
1	001	000
2	010	001
3	011	011
4	100	010
5	101	110
•	•	•
•	•	•
•	•	•

Using binary representation, words of one letter length difference (e.g., 3 and 4) will provide 3 conflict bits while words of many letter length differences e.g., 1 and 5 would produce only one conflict bit. A better representation would retain in the conflict bits the closeness of words of about the same length. (Tenczar and Golden, 1972, p. 7).

The "first character" field is set as follows:

The first letter of a word is perhaps the most dominant. We store information on this character by setting a bit to separate consonants and vowels. Then we map somewhat ambiguously, the specific consonant or vowel onto a few more bits. Although ambiguous mappings lose information, in this case the loss is only partial since the first letter will also set bits in some of the following fields. Here we see an advantage of using redundant criteria rather than a few independent measures. (*ibid.*, p. 7).

The bits in the "letter content" field are set as follows:

The letters of words independent of their order in the word contain information. The reader should have little difficulty in finding a word made up only of one or more occurrences of each of the letters–s, m, p, i–and even less difficulty in finding a word not so restricted. One need not use 26 bits for the letters of the alphabet but can map ambiguously using pairings based on phonetics and data from studies of letter usage in particular languages. For example, the word Mississippi can be mapped into the letter content field as

0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1
E T A O N I S R H L D C U P F M
Q V G Y J K B W X Z

> Here the mapping rule is partially based on letter frequencies in written English. (*ibid.*, pp. 7-8)

The bits in the "letter order" field are set in the following fashion:

> We have found it useful to set bits dependent upon all the digraphs (adjacent letter pairs) present in the word. One possible scheme is illustrated:

SAMPLE WORK	P I E C E
INTERNAL REPRESENTATION	16 09 05 03 05
(a=1, b=2, etc)	
SUM OF DIGRAPH (modulo 10)	5 4 8 8
CONTENTS BITS	0001100100 (10-bit field)

> The value of the digraph sum is used to set bits in the letter order field (the sum is done modulo the number of bits in the field). Note that the frequent spelling error of letter inversion (e.g., ei for ie) will maintain the same digraph sum and thus this information will be passed to the conflict word. (*ibid.*, pp. 8-9)

The bits in the "syllabic pronounciation" field are set as follows:

> The number and sound of syllables in a word are difficult to capture by algorithm. Our trivial but partially successful method is to use the consonant-vowel pairs in a word. Thus California becomes - ca li fo ni - This is about the right number of syllables and pronunciation. Of course, counterexamples can be given to make the scheme look foolish. But we think this pronunciation-syllabification has enough merit to warrant a field of bits. Again, one can set these bits using the consonant-vowel digraph sum and phonetic pairing may be included where desired. (*ibid.*, p. 9)

Tenczar and Golden also make the following general remark about this procedure:

> Several overall characteristics of this mapping algorithm should be noted. First, additional word properties can be easily added to the scheme without major overhaul. One need only add another bit field for each proposed new property. Second, the length and first char-

> acter field at the "top" permits one to order an author's vocabulary numerically yet retain the relative closeness of similar words. A binary chop search can thus be performed on the author's vocabulary and, if no match is found, one is left in an area in which misspelling matches can occur. Third, the exact bit coding schemes and the field lengths are easily varied so one can heed the information theory dictum which states that in a coding scheme one should strive to have a probability of 0.5 of finding any given bit set. (*ibid.*, p. 9)

The total operation of the "misspelled word detector" can be described as follows:

i) The PLATO lesson designer types in the list of correct words he expects students might use, and a separate list of wrong words that students might use; call these words ${}^1B, {}^2B, {}^3B, \ldots$;

ii) The Golden-Tenczar algorithm maps each of these into the symbolic "attribute" words ${}^1B', {}^2B', {}^3B', \ldots$ (note that this is done once and for all at the start of a lesson, and doesn't slow down PLATO's response to individual student inputs);

iii) The student inputs words ${}^1A, {}^2A, {}^3A, \ldots$;

iv) The Tenczar-Golden algorithm maps these into "attribute" words ${}^1A', {}^2A', {}^3A', \ldots$;

v) A "conflict word" is created from discrepancies ${}^1A_1'$ vs. ${}^1B_1'$, ${}^1A_2'$ vs. ${}^1B_2'$, etc., comparing each bit successively in ${}^nA'$ with ${}^nB'$;

vi) The criterion for saying that a student word is a misspelling of an author word is simple: fewer than 7 conflict bits!

The algorithm has been tested for its ability to recognize misspellings from a popular dictionary of common misspellings; the 41-bit algorithm correctly recognized over 95% of the words. (*ibid.*, p. 12)

To test that the algorithm does not match words too freely, pairs of successive words (in lexicographic order) from an ordinary dictionary were compared; less than 14% were accepted as one being a misspelling of the other.

In all, over 2.5 billion test pairs have been compared, and in general it *is* true, if the Tenczar-Golden algorithm declares one word to be a misspelling of a second, a human observer will make the same judgment.

A special virtue of the Tenczar-Golden algorithm is that author words can be mapped into their corresponding attribute words, and then *these later* can be put into lexicographic order, making future comparisons of student inputs possible by a very efficient "binary chop" method, which—when it fails to produce a perfect match—leaves you *at the precise point in the list where a misspelling comparison should be made.*

Sentence meaning recognition

Another Tenczar-Golden procedure allows surprising freedom in pro-

gramming PLATO to respond to student inputs in the form of natural language. (It should, however, be emphasized that the actual vocabulary involved is small, and the universe being talked about is quite narrowly defined. The student, however, usually feels no awareness of constraints, as we shall see in some examples below. After all, he is in a context where there are relatively few things he might wish to say!)

When the PLATO author is designing a lesson, he prepares a list of "ignored words" which will, in fact, be ignored in judging that particular student response. Such a list often looks something like this:

IGNORED WORDS < a, of, for, the, especially, somewhat, her, did, . . . >

For each "important word" he lists the word itself, and also its likely synonyms:

IMPORTANT WORDS (doctor, doc, surgeon, Dr., physician, pathologist)
(patient, woman, female, lady, girl, person)
(visit, see, consult, interview, call)
(lately, recently, yesterday)

If the author has stored an expected student input of the form:

(A) Did the patient see a doctor recently?

This algorithm will automatically accept as (A) a student response if it matches (A) exactly, or if it has some form such as

Did patient see doctor yesterday?
Did the woman consult her physician this week?

and so on. (Tenczar and Golden, 1972, pp. 13-22)

Response to natural language

Now that we have seen how PLATO is programmed to respond to seemingly "ordinary English," provided the domain of discourse is sufficiently circumscribed, it might be instructive to consider an actual example, from a chemistry lesson in qualitative analysis authored by Stanley G. Smith. The task is for the student to identify an unknown, by asking a sequence of questions which PLATO answers. Here is a piece of an actual dialogue:

Student: What color is it.
PLATO: The solid is white and the liquid is colorless.
Student: What is its melting point
PLATO: The melting point is 19-21°.
Student: What does the nmr look like

PLATO: This is the 60 MHz proton nmr spectrum. For a table of chemical shifts, press DATA. (The spectrum is shown on the screen.)
Student: ir
PLATO: This is the ir spectrum (The infrared spectrum is displayed on the screen.)
Student: What is the unknown soluble in
PLATO: Please provide more information.
Student: Is it soluble in H_2SO_4
PLATO: The unknown is soluble in cold conc. H_2SO_4.
Student: Elemental Analysis
PLATO: We can give you results from a sodium fusion. To try it press LAB.
Student: Picric Acid Derivative
PLATO: Please forgive.... The following words are NOT in my vocabulary. . .picric.
Student: dnp
PLATO: The unknown reacts with 2,4-dinitrophenylhydrazine to form a new substance with a melting point of 249-251°.
Student: Where does it boil
PLATO: The boiling point is 195-200° at 760.
Student: Permangenate Test
PLATO: The unknown reacts slowly with $KMnO_4$
Student: Tin and HCl
PLATO: Please forgive.... The following words are NOT in my vocabulary. . .Tin.
Student: Tollens
PLATO: There is no reaction with Tollen's reagent.
The unknown is
Student: Acetphenone
PLATO: SP
Student: Acetophenone
PLATO: OK
For a new unknown press NEXT. (from Bitzer, *et al.*, 1973, p. 5)

Bitzer *et al.*, comment on this dialogue as follows:

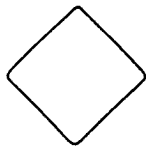
> Shown is a student's misspelled but proper identification. The computer recognizes that the student's answer is correct but that the spelling is inadequate. Spelling algorithms such as this must be a basic part of any educational computer system for, as in this case, if the student were merely told his response is wrong, he could easily spend a great deal of time on the wrong track. (Bitzer, *et al.*, p. 5)

Figure recognition

In a sequence of geometry lessons authored by J. Richard Dennis, students

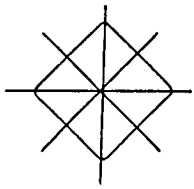
are asked to draw a variety of figures. Dennis has so programmed PLATO that it copes with variations in size, location, and rotation of student inputs, as shown in the following sequence:

PLATO: Draw a quadrilateral with only two lines of symmetry.
Student: [by manipulating a cursor, draws]



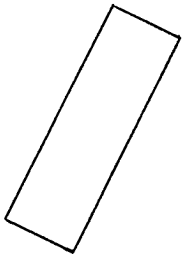
[and then the student pushes a key to ask for PLATO's response to this input, getting:]

PLATO: [inserts lines of symmetry:]

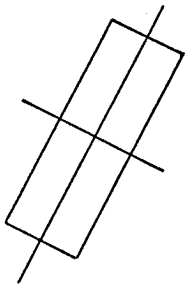


[and writes]
No, your figure has four lines of symmetry. Please try again.

Student: [again manipulating the cursor, draws]



PLATO: [again inserting lines of symmetry]



[also writes]
Good! Your figure has symmetry lines that do not go through vertices.

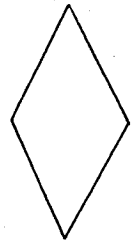
Student: [presses the NEXT key]

PLATO: Now draw a quadrilateral with only two lines of symmetry that DO go through vertices.

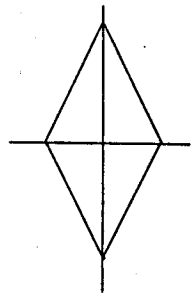
Student: [uses cursor to draw]

PLATO: You drew that kind of figure before. Are you trying to fool me?

Student: [draws]



PLATO: [inserts lines of symmetry]



[and says]
Very good!!

(Bitzer, Sherwood, and Tenczar, 1973, pp. 30-31)

ELEMENTARY MATHEMATICS COURSEWARE

As indicated earlier, the authors are now engaged in the creation of appropriate courseware to help students in grades 4 through 9 to learn mathematics. For the remainder of this paper we will look at aspects of this courseware that seem relevant to artificial intelligence. The basic goal of the courseware is to program PLATO so that if fits into the ecology of the classroom the way that an intelligent human being would want it to. This obviously includes helping students to learn mathematics in various narrower senses, but has also a broader and

less precise meaning which we hope will become somewhat clearer as we look at examples.

The session selector

Observation of human teachers indicates that they often "compose" a day's lesson so that there is a kind of harmony or balance to it. If some part of the lesson calls for very careful attention to intricate minute detail, then another part of the lesson may be of a different, broader, less "fussy" nature. If part deals with some matters that students are finding uncongenial, then another part may be something they enjoy. If part is difficult, another part may be easier; if part is review, another part may involve the first presentation of some new ideas.

There is also a balance between tasks which the teacher selects, and even methods of attack suggested by the teacher, balanced against tasks or methods suggested by the students.

We obviously want PLATO to compose the day's lesson in somewhat the same way. To achieve this we have a lesson-assignment mechanism, programmed by Glen Polin, that divides the prescribed total lesson time (presently set at about 30 minutes in most cases) into three "slots." The second of the three slots is, from a curriculum point of view, the most fundamental. It is computed from a curriculum chart, and from data on previous student performance, and represents the main "new material" for the day's work, for that particular student. The second slot assignment is computed first, after which an assignment for Slot I is computed, often consisting of earlier content material that should be reviewed briefly since it will be needed for the work in Slot II. Finally, a "menu" of possible activities is computed, for each individual student, and displayed to the student at the start of Slot III, which is a "student choice" slot, usually including a number of mathematical games that have proved popular with the students.

In order to make sessions last approximately 30 minutes—and the "approximately" is important, since a student is not interrupted in the middle of a task or unit—PLATO uses two sets of numbers: for each unit, the average length of time the unit has taken previous students; and, for each student, a "personal speed factor" that compares his individual average speed with the average of all students. Thus, if Johnny V. takes, on the average, 75% as much time to complete a unit as most students take (averaged), and if Unit 35 has taken, on the average, 8 minutes, PLATO would project a 6 minute expected time for Johnny V. to complete Unit 35.

The PLATO elementary mathematics material is organized into three strands: a strand on the arithmetic of whole numbers, a strand on the arithmetic of fractions and rational numbers, and a strand on algebra and analytic geometry. Each strand is subdivided into modules that are sequential. A classroom teacher can prescribe the sequences for each student to work on; for example, she can say that Tommy W. should work on fractions on Mondays, Wednesdays, and Fridays, beginning with Module 3 on equivalent fractions, and on Tuesdays and

Thursdays, for variety, he should work on the algebra strand, starting at the beginning. The automatic lesson-assignment mechanism, which is known as the SESSION SELECTOR, would accept this teacher input, comply with it, and provide daily lessons for Tommy W. that were consistent with the teacher's request.

Teaching method

While the key words for PLATO courseware are surely variety and diversity, there is one central idea that underlies much of the elementary mathematics courseware. This is the idea of the *paradigm* presentation of new concepts, namely: do not *describe* a new concept, do not *define* it–neither approach will succeed with most students–but *illustrate* it. Create some simple, memorable *instance*, perhaps (if necessary) using some application or social situation, conceivably (if necessary) even a fanciful one. Future instances will be recognized by their similarity to the original "defining" instance (Davis, 1972). (We suspect this may be far more than a valuable teaching strategy; it may come close to the nature of much human information processing: new inputs may be mapped into some stored "paradigm" or "metaphor".) To give one instance, the introduction of negative numbers is achieved by a *paradigm* presentation involving a bag partially filled with pebbles, a separate pile of loose pebbles, and someone (Terry, say) who says "Go!" (thereby marking a reference point in time), after which, say, we put 4 more pebbles into the bag, writing

$$4,$$

whereupon we remove 5 pebbles from the bag, writing

$$4\text{-}5,$$

and ask: "Are there more pebbles in the bag now than there were when Terry said 'Go.'?" Since there are fewer pebbles in the bag–indeed, one less–we write

$$4 - 5 = {}^{-}1.$$

(cf. Davis, 1967, Chapter 4). The extension of this approach should be apparent.

The nature of arithmetic

Balancing arguments for and against various approaches, we have concluded that for young children it is desirable for them to regard mathematics as consisting of "statements" or "stories" about the real world. Hence our courseware stresses the ability to go back and forth between any such statement

- say,

$$8 \div 2 = 4$$

- and a "real" situation that would match it, such as "I had 8 cookies, and I

shared them equally with my sister." Given such a word story (perhaps told partly via pictures), we want students to be able to write the corresponding "mathematical story"

$$8 \div 2 = 4$$

(or, conceivably, $8 - 4 = 4$, etc.),

and, conversely, given the "mathematical story"

$$8 \div 2 = 4,$$

we want students to be able to write an appropriate matching story in words or in pictures. While some of the sentence interpreting problems are quite beyond our present capacity, PLATO's visual display capabilities are very helpful in implementing this approach.

Kinds of math courseware

If we try to sort out the elementary mathematics courseware, at least roughly, according to the categories suggested by Resnick (see above), the picture might look like this:

1. Sequences of "bite-sized" chunks of mathematics: this category implies that the computer is individualizing *only* to the extent of perhaps using past student performance to determine the difficulty of the next task, or the pace of moving along the curriculum strand. This might be a fair description of some short sequences of the PLATO mathematics courseware, but probably fails to describe the larger forms of individualization that are employed.
2. Individualization by additional student attributes, such as cognitive style.
   a. We have already indicated how daily sessions are calculated using a personal "time factor" for each student;
   b. Within many lessons there is a minimum amount of material that the student is required to cover; however, as soon as he has completed this much material, he has complete control over when he exits from the lesson—and if he exits before his 30-minute session his over, PLATO will present him with whatever has been computed to come next (very likely a "choice page" for the beginning of Slot III).

      This in effect turns one aspect of student learning style directly back to student control, as an alternative to trying to develop an algorithm to compute appropriate differences. In later examples, we shall give further consideration to this method of finessing certain computations.
3. Individualization by trying to recognize, and interact with, the student's cognitive structure.
   a. Of course, at its simplest level, this includes quite standard programming for PLATO responses to student errors; as, for instance, in a sequence such as:

PLATO: 8 x 8 =
Student: 8 x 8 = 16
PLATO: Did you mean to answer 8 x 8 = ,
or 8 + 8 = ?

b. PLATO courseware regularly carries this a great deal further. An effort is made to identify every likely "wrong answer," to identify the error in student thinking that probably led to this answer error, and to provide a PLATO response that will help correct the error in thinking.

This, in a math lesson called HOP, written by Bonnie Anderson, at each point a long list of wrong answers has been programmed in, and each triggers an appropriate remediation sequence.

c. In the following section we shall see a further elaboration of this approach, as carried out in Resnick's thesis.

Some assorted examples involving A.I.

Categories of student strategy in WEST (from Resnick, 1975).

WEST is a game between two opponents, a "stagecoach" and a "locomotive."

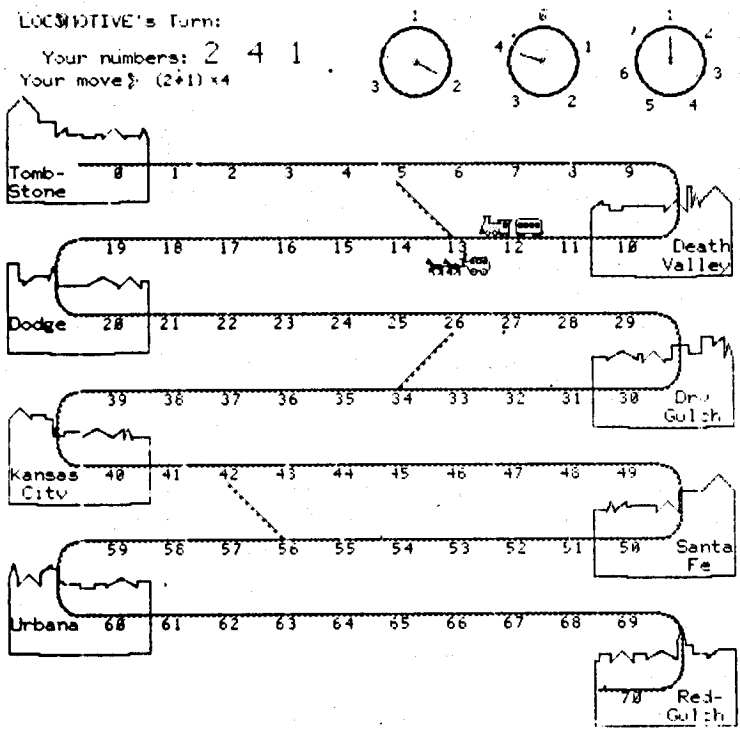


FIG. 4. Screen display from Lesson WEST.

It was written by Bonnie Anderson for the PLATO Elementary Mathematics Project. The game is intended to provide elementary school children with "painless practice" in arithmetic.

The game was chosen for several reasons. It is well designed and programmed so that the player is not frustrated. It is one of the most popular games on the system so it is easy to get players to cooperate. Finally, the game allows for a wide variety of player behavior. I had noticed that despite the varieties of possible behavior, individual players seemed to behave in patterned ways. This observation provided the impetus for further exploration into the nature of this patterning.

Rules

"West" is a game between two opponents, a "stagecoach" and a "locomotive." Each battles to beat its opponent to the goal which is the final position (location "70") on a linear game board. Each move consists of four steps: the spin, the expression, the acceptance of the expression, and the evaluation of the expression.

In step one, the player presses a key to spin three spinners whose values are determined randomly by PLATO (hence they are not under player control). The three spinners have integer values whose range is for the leftmost a = {1,2,3,} for the middle b = {0,1,2,3,4} and for the rightmost c = {1,2,3,4,5,6,7}.

In step two, the player then combines these three spinner values with two of the four arithmetic operators, +, -, x, /, to create an arithmetic expression. Thus, the spinner triplet [a,b,c] can be combined to form axb+c or cxa-b, etc., but not, e.g., bxaxc etc., because in this latter expression only one type of operator is used. Without parentheses, the operations of x and / will be performed before + and -. Thus, a+bxc is *not* equal to (a+b)xc. (This is a frequent source of wrong answers.)

In step three, the expression entered by the player is checked by PLATO for legitimacy. Troublespots are indicated to the player who must re-enter a correct expression in order to continue.

Finally, the player evaluates his chosen expression and enters this value "v" into PLATO. If the value is correct, the player advances "v" spaces forward from his current location on the linear game board. If the value is incorrect, the player loses his turn. When this step is concluded, the opponent begins his move and the cycle repeats. Play continues until the goal is reached.

Certain locations on the game board are designated as special moves: "towns," "shortcuts," and "bumps."

Towns occur on locations that are multiples of 10 (0, 10, 20, etc.). If a player lands on a town, he is automatically advanced ten

spaces to the next town.

Each game contains three shortcuts. If a player lands on the beginning of a shortcut he is automatically advanced to the end of the shortcut. The precise locations of the shortcuts vary from game to game.

If a player lands on the same location as his opponent, the opponent is "bumped" back two towns with the exception that a player cannot be bumped if he is located in a town.

The winning player is the first player to land exactly on location 70. Both players can win (tie) if they get to 70 on successive moves and if the first to win was the first to begin the play. (Resnick, 1975, pp. 5-8)

Now, the lesson WEST serves several different educational goals. It reviews arithmetic in general. It can (but need not) review specific skills, such as the correct use of parentheses. It can serve to get a student started on a whole new train of thought: *maximizing* the value of an expression, or achieving certain special values.

Whether these occur depend largely upon the specific thinking of each child.

Concerning ideal strategies for winning at WEST, Resnick describes the situation as follows:

> The object of the game is to beat your opponent to the goal. A good strategy is generally to get as far as possible on a given move. The largest numerical result from a given set of spinners is determined by multiplying the largest number (L) by the middle-sized number (M) and adding the smallest (S), i.e., LxM+S (or some equivalent alternates of this expression, i.e., LxM+S, MxL+S, S+LxM, S+MxL). If parentheses are used, the largest numerical result is derived from (S+M)xL or some alternates. This usually gives a larger result than LxM+S. If the spinner values contain a 1 and 0, the largest result is: L+1-0.
>
> Another means of getting large results is by landing on a special move: a town or a shortcut. Bumping may not advance the player maximally but it may increase the distance between him and his opponent.
>
> To win the game, a player must land exactly on location 70. An overshoot results in a loss of turn. Thus, the usual strategy to get as far as possible is not appropriate near the goal, and other strategies ("end game") may be adopted. (*ibid.*, pp. 8-9)

In a usual classroom situation, a child who had not thought to use parentheses can get the idea from observing some other child do so. But, given computer-assisted individualization, this process of "picking up good ideas from others" may not occur–unless we help it along! This is what Resnick set out to do.

First, she used a computerized data collection system to preserve and analyze all PLATO stimuli (such as spinner outputs), and all student inputs to WEST from 34 students in grades 4, 5, and 6;

On this basis (neglecting, in this brief summary, some refinements that were made, as in a separate analysis of end-game strategy), she identified three basic categories of student strategy:

1. Invariance of spinner order and operation order.

Since the spinners present the 3 numbers in a left-right sequence, it is possible for a child to use the 3 numbers in this same left-right sequential order—and many do.

This is "invariance of spinner order."

Concerning the arithmetic operations +, -, ÷, x, PLATO does not present these to the student at each turn, so that mimicking PLATO's order is not possible; it is, however, possible to continue to repeat whatever pattern one happens to use first, and many children do this. Resnick calls this "preserving operation pattern." Consider, for example, this data from a game played by a child named Bridget:

Spinner	Move
1,3,5	1x3+5
1,2,4	1x2+4
2,4,6	2x4+6
3,4,2	3x4+2
2,0,5	2x0+5
3,0,2	3x0+2
3,3,4	3x3+4

(*ibid.*, p. 41)

By familiar methods, Resnick sets bits so that, for the i^{th} move, R(i) = 1 if spinner order is preserved, and R(i) = 0 otherwise. P(i) = 1 if the operation pattern is preserved, and P(i) = 0 otherwise. Bridget's play yields the table

Move #	R	P
1	1	1
2	1	1
3	1	1
4	1	1
5	1	1
6	1	1
7	1	1

(*ibid.*, p. 42)

The product

$$\pi = R(1)P(1)R(2)P(2)\ldots R(n)P(n)$$

(where n is the number of turns) will equal 1 if and only if both spinner and operation patterns are preserved. If $\pi = 1$, the student is classified in category A.

2. Getting the largest possible number.

A better, though imperfect, strategy is to arrange spinner numbers and operations so as to produce the maximum value. Thus a spinner output of 1,5,3 would be used to make the expression

$$5 \times 3 + 1$$

if a student does not use parentheses, or

$$(3+1) \times 5$$

if the student does use parentheses. Students who use this strategy are classified into category B.

3. 3. But there is a still better strategy: maximize your lead over your opponent. This requires flexibility: use the largest expression value if that is optimal, or land on a town if that is optimal, or take a shortcut if that is optimal, or bump your opponent back if that is optimal. (We ignore the probabilistic issue of seeking the safety of a town so that the opponent can't bump you.) Students who use a reasonable approximation of this strategy are classified into category C.

Now, what would one *wish* a computer to do? If a child has never thought to use parentheses, you'd like the computer to use them, to suggest this possibility. If a child has never used shortcuts, again you'd like the computer to use this strategy (when possible) on its turn. Or, given spinner output

$$1, 0, 1$$

if your opponent is one space behind you, the optimal strategy may be to make the expression

$$0 - (1 \div 1)$$

and bump him back, while losing one space yourself. Few children think of this; again, PLATO can suggest this possibility by using it.

Resnick's thesis identifies, in this way, a student's cognitive structure. One thus knows *what step* might be a reasonable next step for this student to take. (Unfortunately, preliminary data suggests that many children focus so tightly on their *own* play that they fail to notice the strategies that PLATO uses, and attribute PLATO wins to the fact that PLATO cheats by manipulating spinner output (which does not occur).

We have yet to make this type of analysis for other situations in PLATO courseware nor (apparently) have we found the best way to show a child some new possibility that he isn't using; nonetheless, a real possibility clearly exists, to

develop CAI materials that fall clearly into Resnick's third category, of recognizing the child's cognitive structure, and interacting with it so as to promote its further development. To a Piagetian, the further development of cognitive structure is what learning is all about, and aiding this process is what teaching is all about.

The "finesse" strategy

For the most part, present PLATO elementary mathematics courseware attempts to get "human-like" behavior from the computer not by A.I. or algorithmic means, but by returning the task to human beings. This is especially true of our fourth category of CAI—where our concern is for how the ecology of the classroom is modified by PLATO's presence.

Consider one example: in non-computer classrooms, a child may do something clever and original, and proudly display it to others. Children get ideas from one another. Children set up their own friendly competitions. PLATO, too, provides for this.

A PLATO lesson created by Kibbey and Dugdale asks a child to color in part of a rectangle, predicting in advance what fraction of the rectangle he will color. In Figure 5 the task was to color in 3/5 of the rectangle, but the student has colored in 11/20. PLATO points out the error.

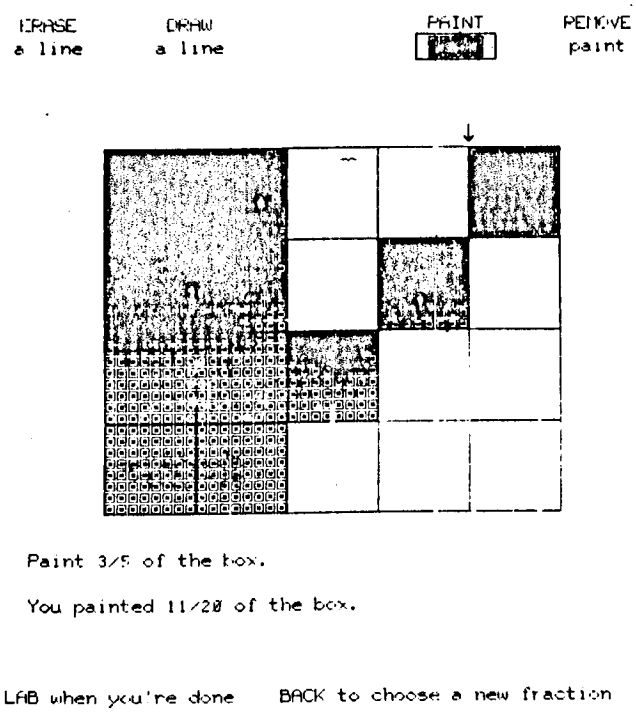


FIG. 5. The PLATO display panel points out an error.

In Figure 6 we see the PLATO screen after Sharon has predicted that she will color in three fifths of the rectangle, and has successfully done so.

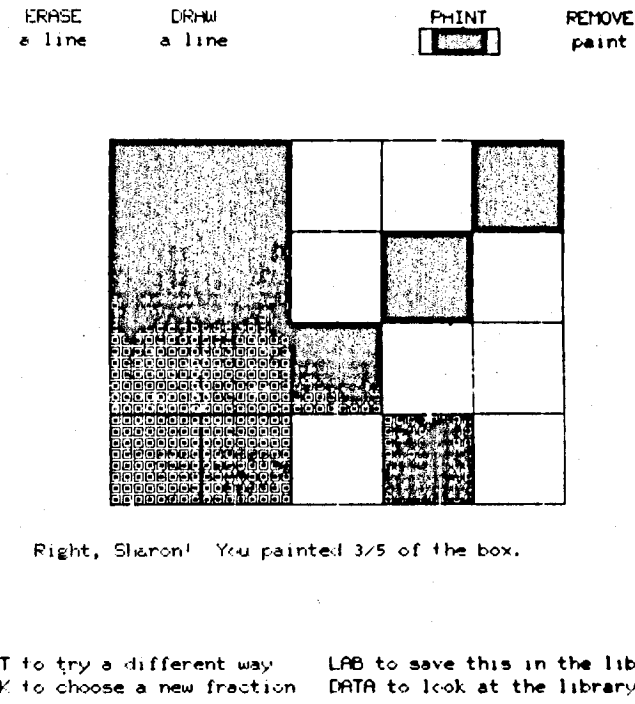


FIG. 6. The PLATO display panel after Sharon has predicted that she will color in three fifths of the rectangle and has successfully done so.

After a child has colored in some rectangles by himself, two things happen:

i) PLATO allows him to *save* two of them in a "paintings library";
ii) he can access this library and look at all of the paintings saved by other children.

After looking at the library, the child can make further paintings of his own, and he can replace his own "saved" paintings with newer ones if he wishes. In this way, the following sequence of paintings was generated:

This is how lovetta h painted 1/2 of the box.

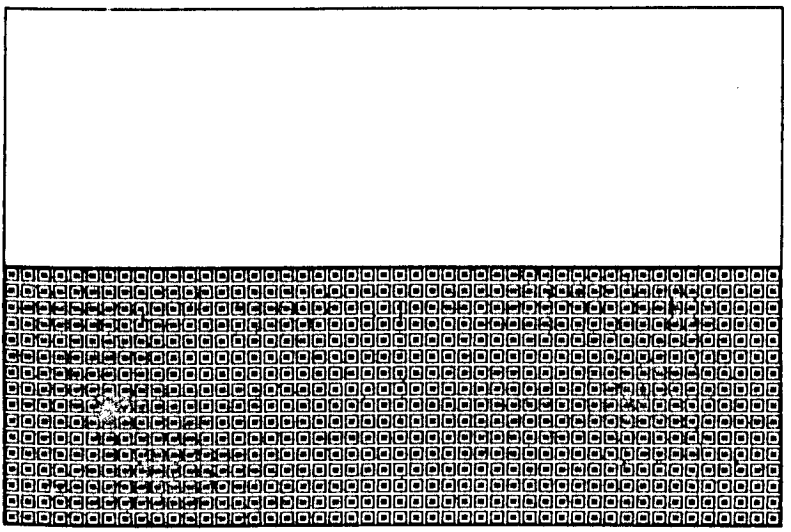


FIG. 7. The PLATO display panel, showing a painting from the "paintings Library."

This is how varaidzo m painted 1/4 of the box.

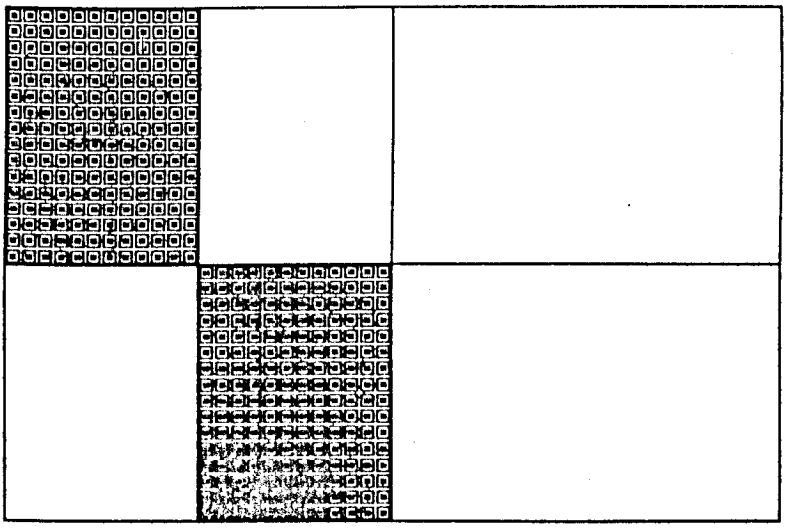


FIG. 8. Another painting from the "paintings Library."

This is how frederic m painted 1/2 of the box.

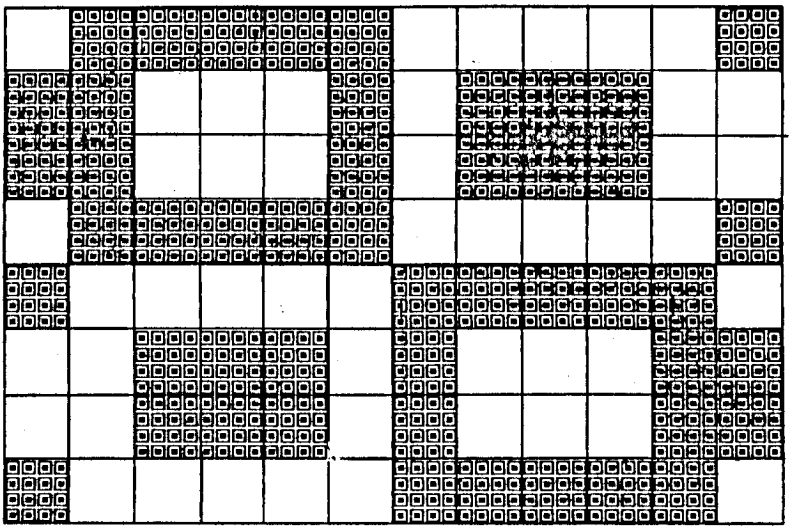


FIG. 9. Frederic M.'s contribution.

This is how fred s painted 1/2 of the box.

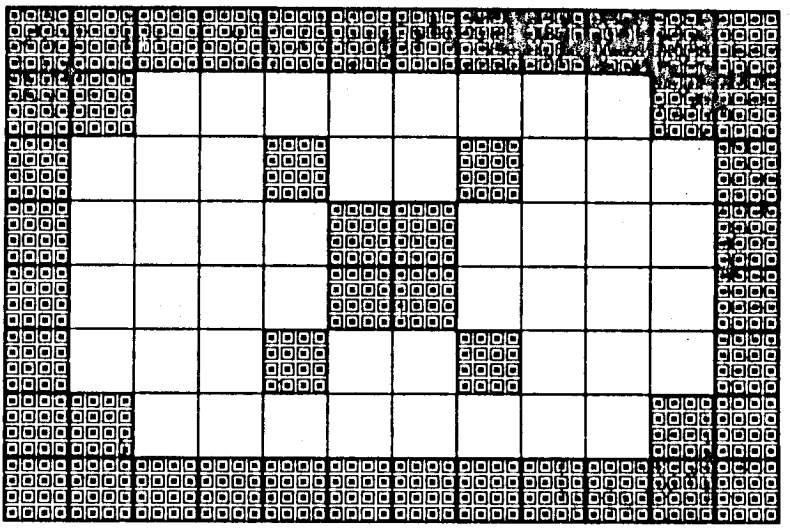


FIG. 10. Fred S. achieves a new level of artistic elegance, while getting exactly the 1/2 he was aiming for.

This is how lawston t painted 5/9 of the box.

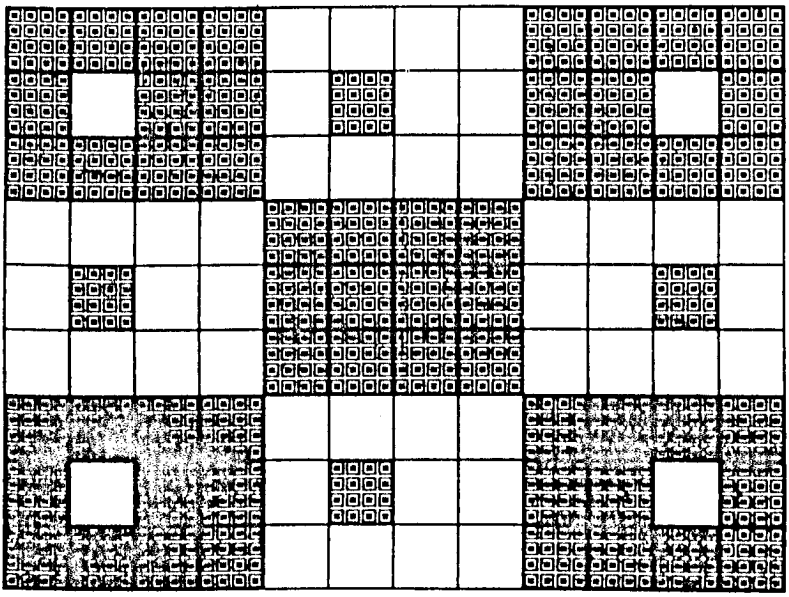


FIG. 11. Lawston T. suggests a new possibility for easy estimating of the fractional part.

This is how fred d painted 1/2 of the box.

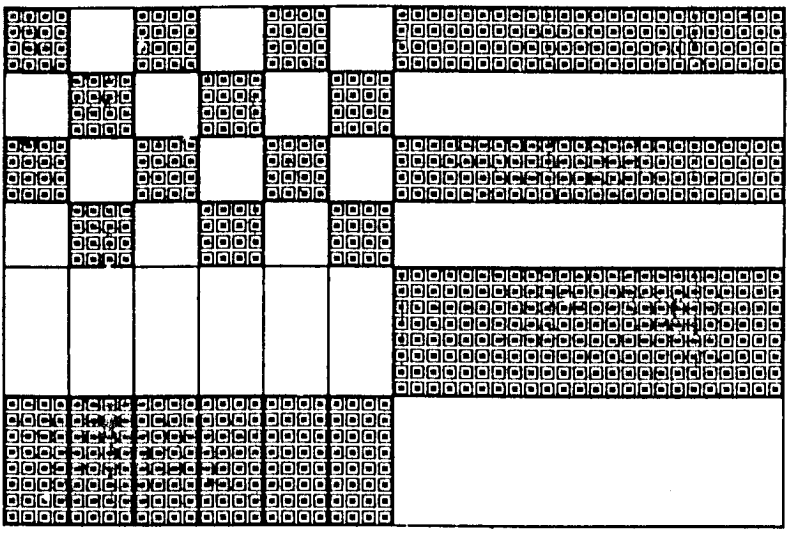


FIG. 12. . . .but the main competition, for the time being, focuses on one-half.

This is how cynthia s painted 1/2 of the box.

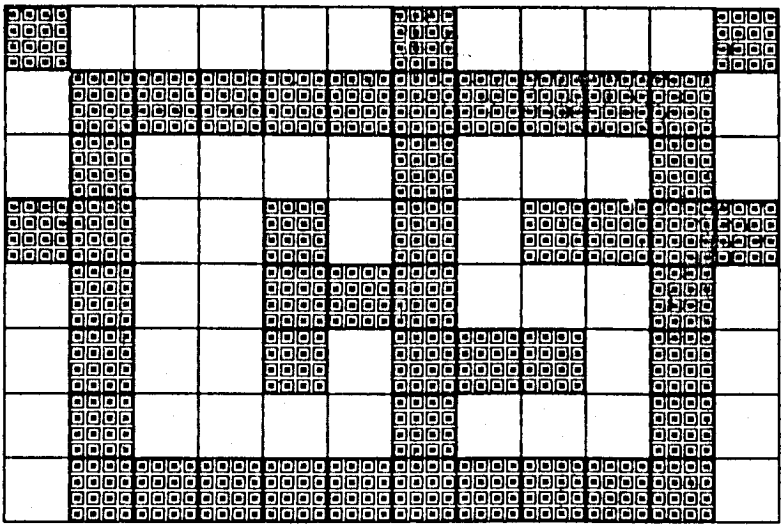


FIG. 13. Cynthia S. makes a break-through!

This is how susan h painted 1/2 of the box.

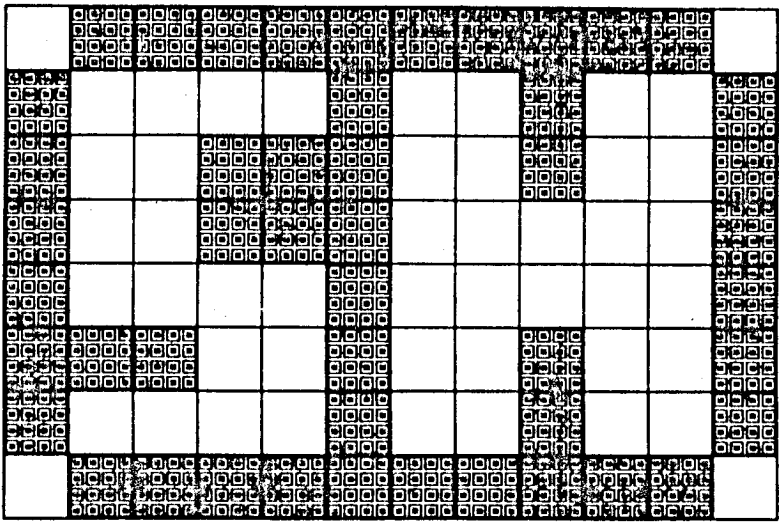


FIG. 14. . . .which Susan H. pursues.

This is how asa a painted 1/2 of the box.

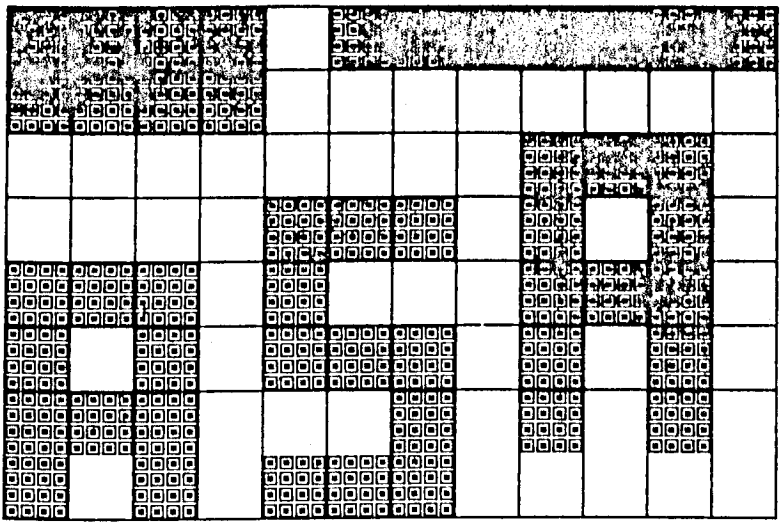


FIG. 15. . . .but why stop with initials? Here is Asa A.'s contribution.

This is how ricky s painted 1/2 of the box.

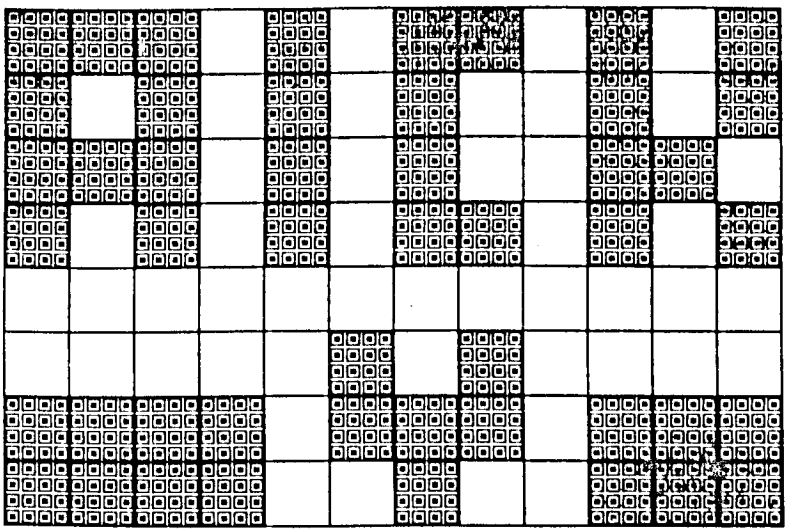


FIG. 16. . . .and Ricky's. . .

This is how douglas s painted 1/2 of the box.

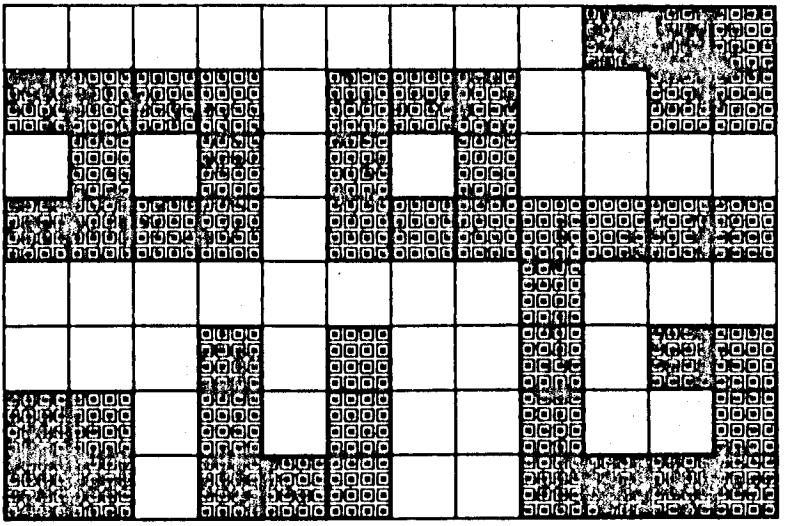


FIG. 17. . . .and a particularly elegant one by Doug.

This is how margaret s painted 2/7 of the box.

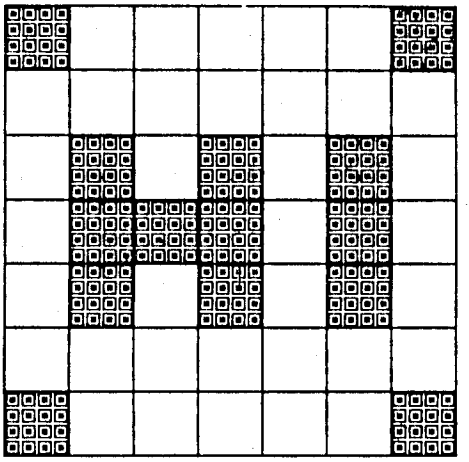


FIG. 18. For some reason, the students switch to "Hi!? (Perhaps because it's a fairer competition if everyone uses the same word?)

This is how raymond h painted 1/2 of the box.

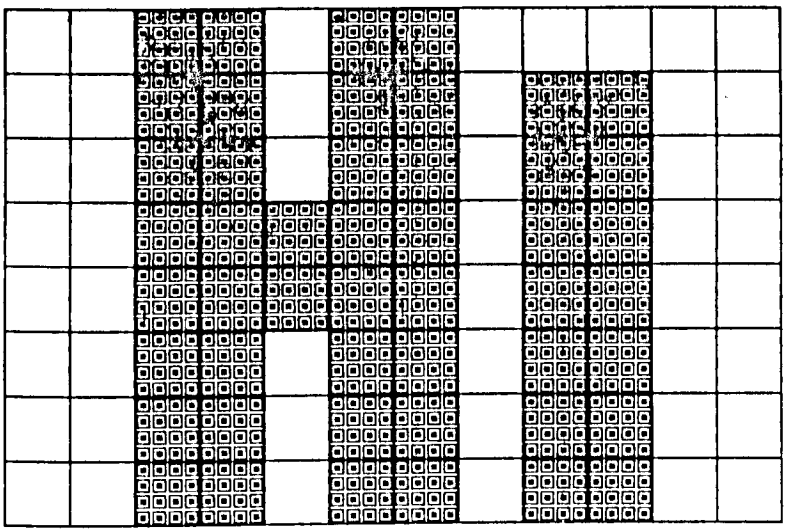


FIG. 19. It's a more elegant solution if you don't have anything left over.

This is how jeffrey s painted 1/2 of the box.

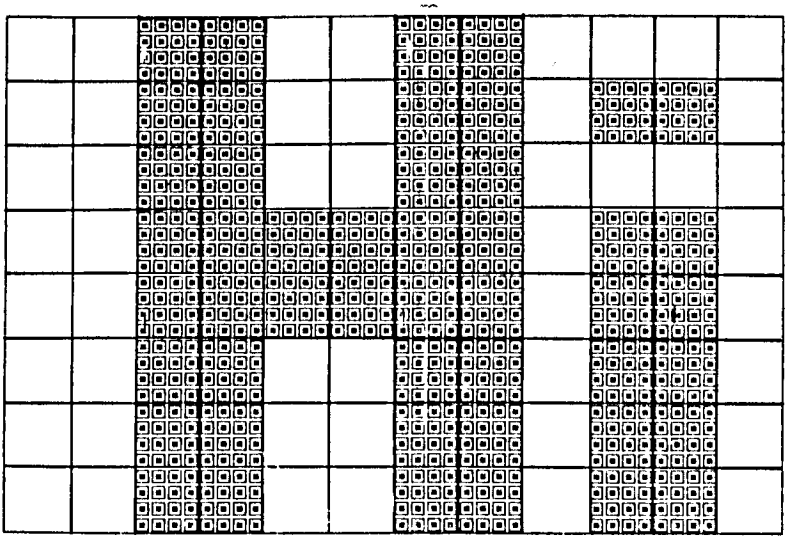


FIG. 20. Variety is still possible.

This is how derek k painted 1/2 of the box.

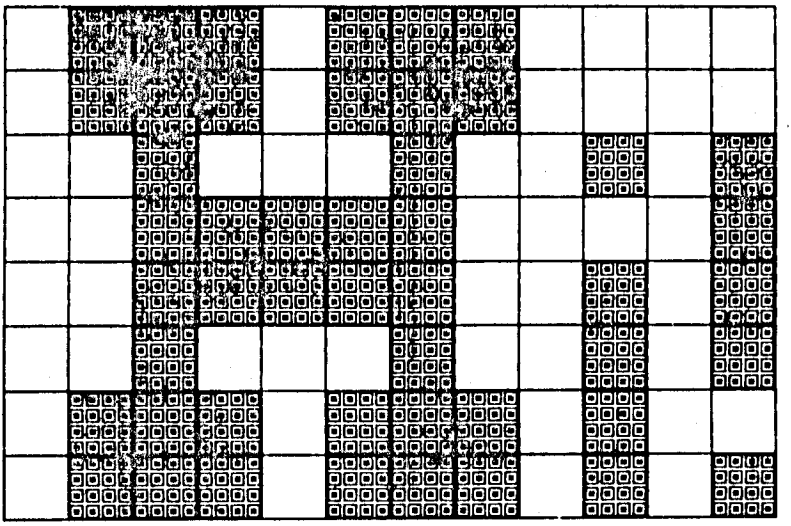


FIG. 21. . . .and it's still exactly half!

This is how theodore p painted 1/2 of the box.

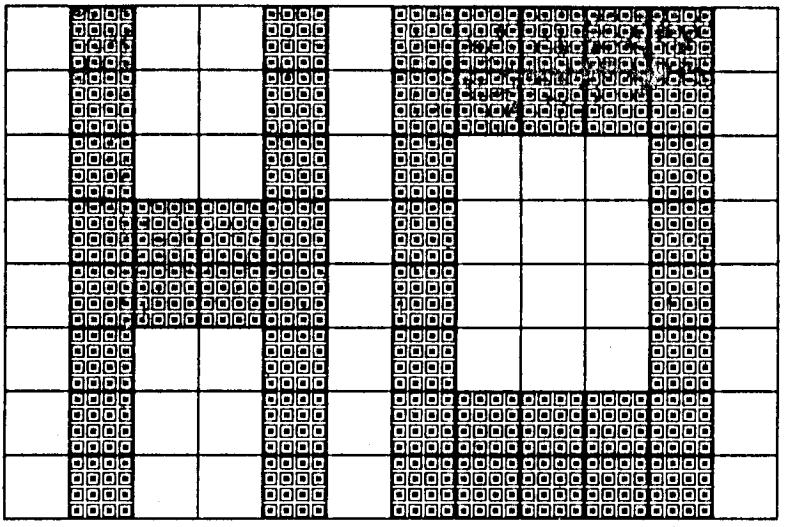


FIG. 22. Time for a change.

This is how jeffrey s painted 1/2 of the box.

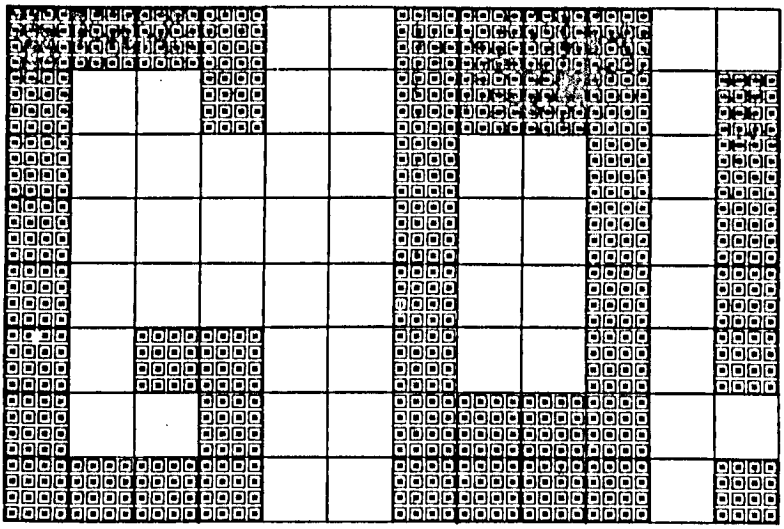


FIG. 23. . . .and still exactly half!

This is how jim n painted 1/2 of the box.

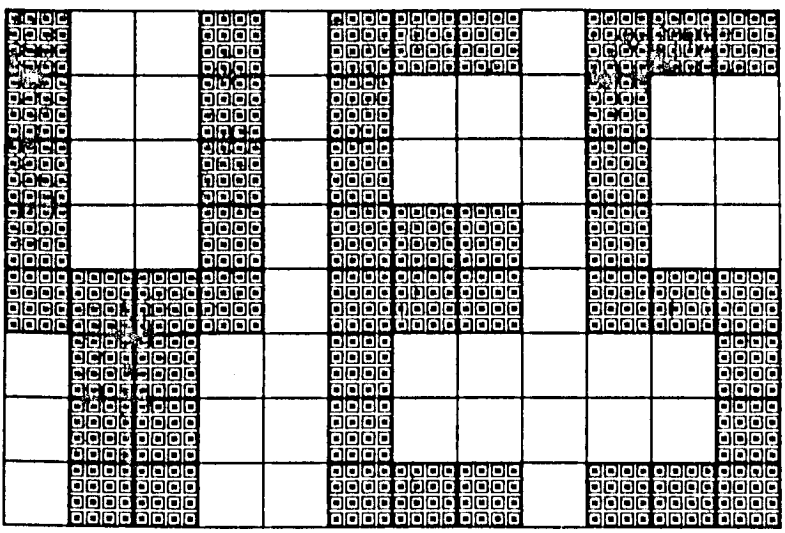


FIG. 24. . . .affirmation. . .

This is how david b painted 1/2 of the box.

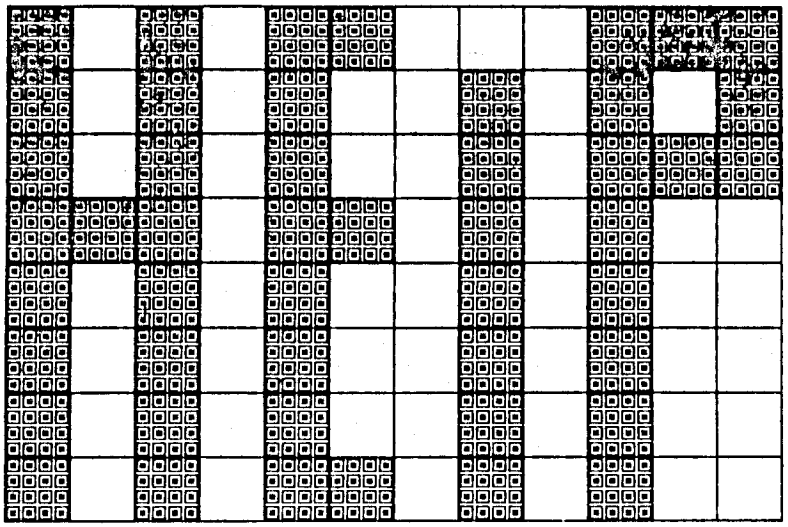


FIG. 25. . . .and commentary. . .

From our "human," or "classroom ecology" point of view, we want to make sure that the clever and the elegant and the original are honored: we want to increase, rather than diminish, positive forms of social interaction; and we want a student to grow in an environment where his personal learning is part of, borrows from, and contributes to the general culture. PLATO handles this very nicely!

SUMMARY

PLATO is a large CAI installation with a "practical" mission to provide instruction to schools and universities. If any original contributions to A.I. have occurred they were fortunate accidents, and we are aware of none. However, rather mundane applications of A.I. abound within PLATO software and courseware, and much more is possible in the future (and likely, if the pressures to provide basic instruction gradually abate), and some of these possibilities can begin to be discerned. For the present, some of the more "human-like" performance is achieved, not by A.I., but by turning part of the task back to human beings. An example is the recognition of ingenuity in the ways children color in a specified fraction of a rectangle.

BIBLIOGRAPHY

Bitzer, Donald L., Sherwood, Bruce Arne, and Tenczar, Paul (1973) *Computer-Based Science Education,* CERL Report X-37.

Davis, Robert B. (1966) *Explorations in Mathematics: A Text for Teachers.* Addison-Wesley Publishing Company.

Davis, Robert B. (1971) The structure of mathematics and the structure of cognitive development. *The Journal of Children's Mathematical Behavior,* 1, 1, pp. 71-99.

Davis, Robert B. (1972) Naive foundations for a theory of mathematics learning. *Learning & The Nature of Mathematics,* ed. William E. Lamon, Science Research Associates, Inc.

Davis, Robert B. (1974) What classroom role should the PLATO computer system play? *Proceedings of the 1974 National Computer Conference,* AFIPS Press, pp. 169-173.

Flavell, John H. (1963) *The Developmental Psychology of Jean Piaget,* Van Nostrand.

Koestler, Arthur (1959) *The Sleepwalkers,* Macmillan, (re-issued in paperback by Grosset and Dunlap, 1963).

Kuhn, Thomas S. (1962) *The Structure of Scientific Revolutions,* University of Chicago Press.

Polanyi, Michael (1958) *Personal Knowledge,* University of Chicago Press.

Resnick, Cecily (1975) *Computational Models of Learners for Computer Assisted Learning,* unpublished Ph.D. thesis, University of Illinois, Urbana.

Tenczar, Paul, and Golden, William (1972) *Spelling, Word, and Concept Recognition,* CERL Report X-35.

20

Three Interactions between AI and Education

Kenneth Kahn

MIT AI Laboratory
Cambridge, Massachusetts

The understanding of intelligence that has arisen from AI research can be applied to education to significantly enhance learning. The LOGO project, described herein, is based on this premise. Extensions of the LOGO experience are proposed that are based on AI in interesting ways. One extension is to encourage children to write simple AI programs. Tools to aid the child in this endeavor are discussed. Another supplement to LOGO is the interaction of the child with AI programs specially designed for this purpose. These programs, in addition to being easily used by children, should be modifiable by children in interesting ways. The specifications of such a program are described.

INTRODUCTION

This paper describes some ideas concerning three different ways in which AI can play a role in education, which are illustrated in Figure 1. Children can both be encouraged to write simple AI programs and to interact with specially designed AI programs. Both of these interactions with AI are guided by the understanding of intelligence and learning that has arisen from AI research. These ideas are within the framework of the LOGO project at MIT. LOGO is concerned with the more general problem of enhancing education by applying the concepts that have come from AI research. A description of how LOGO attempts to do this is described in the next two sections. Following that are some of the author's ideas as to how AI can enhance education in ways that supplement LOGO's.

THE LOGO PHILOSOPHY

One of the main occupations of a child is to learn. Yet, typically no effort is made in schools to encourage the child to learn how to learn, or even what this means. The explicit teaching of general principles of learning, at least without many examples, does not accomplish this task. The theory, to the child, is abstract and difficult to understand—even more difficult to use.

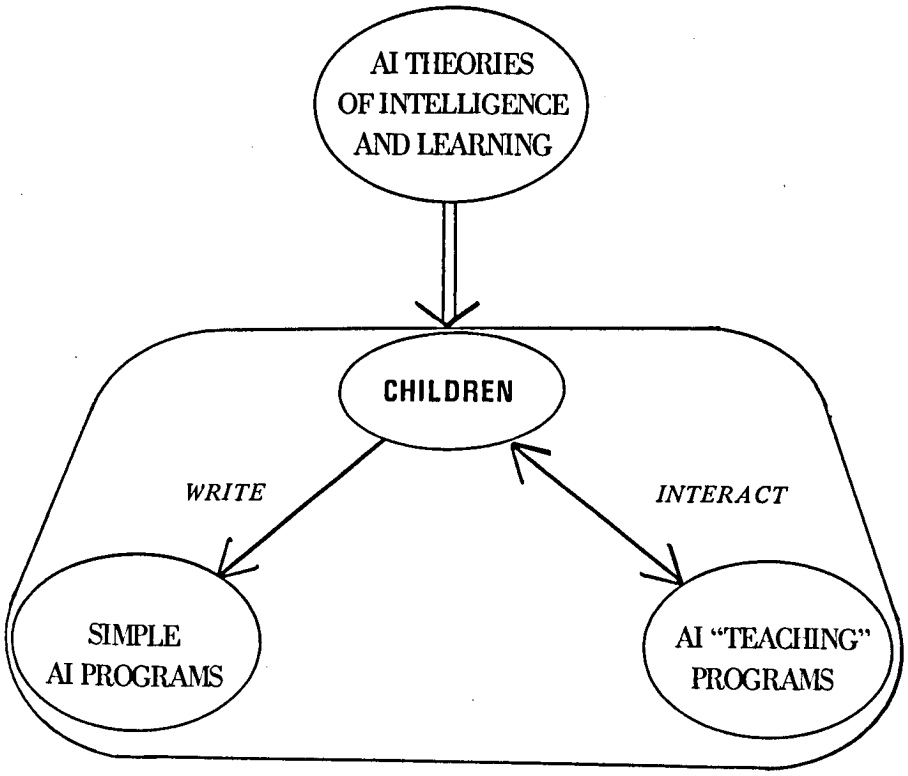


FIG. 1. The three roles of AI in education

The child should, instead, acquire some good "powerful ideas"† for learning and problem solving. The discovery of "powerful ideas" is central to AI research and some have been developed. For instance, the notion of "linearization," or the strategy of attacking the sub-problems of a problem independently, and then dealing with the interactions is a powerful idea. Another one is to view "bugs" in a solution as things to correct and learn from, *not* mistakes that lower a test score. These ideas are typified in Sussman's HACKER (Sussman, 1975). Some other powerful ideas are "naming of concepts", "planning", and "being explicit about one's one thinking". Most important of all the powerful ideas is the notion of a powerful idea itself.

If children could really learn these powerful ideas they would presumably be better students and better problem solvers. But how are they to learn these ideas? What is needed is an environment where these notions surface frequently, where one needs to be explicit about problem solving, an environment where the problems are often found to be intrinsically interesting, an environment where

†As they are called by Seymour Papert

the children constantly use powerful ideas to accomplish their goals. Programming is one such environment. "Linearizing" becomes "implementing subroutines independent of other subroutines". "Naming of concepts" becomes "naming subprocedures and variables". Debugging is an important component of programming.

There are other beneficial effects of choosing programming as an environment which if properly created can help the child become a better student. These are:

1) The experience of working on a long-term project, solving sub-problems and planning as one goes
2) The experience of having explicit control over one's environment
3) The experience of learning about the domain to which the child's programs apply
4) The experience of finding that mathematics can be useful.

Equally important is that children usually find it fun.

The reader should note, however, that there is no claim made here the programmers are any smarter, or any better learners, than other people. It is not the act of programming that encourages one to pick up powerful ideas, rather it is programming well chosen problems, within a language that encourages powerful concepts (such as recursion), and with proper guidance by a teacher.

Computers provide a rich environment for the child to develop concepts of problem solving and learning. Other environments, however, are also useful towards this end. The learning of physical skills, such as circus arts, constitute such an environment. The main thesis is that the *right* verbal description of a physical skill does aid greatly in learning it. Also, interaction of learning in this domain and programming is rich. The child can learn that the powerful ideas learned while programming are useful for more everyday activities.

THE LOGO LANGUAGE

The LOGO language was developed to provide the kind of environment in which children can learn to learn. It is a programming language that is human engineered (or more accurately "child engineered"). The structure of the language, the primitives, and their names were designed to aid in the kind of conceptual thinking described above.

An important part of LOGO is a form of computational geometry called "Turtle Geometry". A turtle is a computational entity with a state consisting of a position and a heading. A turtle accepts commands, telling it move forward or backward, that change its position. It also accepts commands to turn left and right, thereby changing its heading. Turtles are usually realized as physical devices called "floor turtles" or as "cursors" on a display. Turtles also have a pen that can leave a trail as they move.

As an example, let us consider a turtle procedure for drawing a polygon. The procedure expects two inputs, the size of each side and the amount the turtle

should turn after drawing each side. More details are shown in Figure 2.

```
TO POLY :SIZE :ANGLE
10 FORWARD :SIZE
20 RIGHT :ANGLE
30 POLY :SIZE :ANGLE
END
```

FIG. 2a. Turtle procedure for drawing a polygon

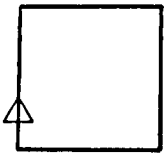


FIG. 2b. Picture produced by the call POLY 100 90

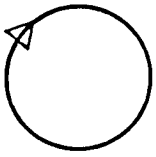


FIG. 2c. Picture produced by the call POLY 1 1

FIG. 2. An example of Turtle Geometry

Children can prove Turtle Geometry theorems that are useful, not only for proving more theorems, but for writing programs. A theorem called the "Total Turtle Trip Theorem" is an example. It states that if a turtle takes a trip and ends up in the same state (position and heading) in which began, then the total amount turned right (minus that turned left) is a multiple of 360 degrees. Children usually understand the intuition of the proof when asked to think about a broken turtle that could not move, only turn. This theorem is very useful in writing state independent subprocedures.

The previous sections describe the LOGO Laboratory as background for what follows. The rest of this paper deals with ideas and research of the author that is consistent with the LOGO framework, but relies more heavily on AI research.

AI FOR CHILDREN

There are several good reasons why children should write and interact with AI programs, a few of which follow:

1) Children are encouraged to think explicitly about how they solve problems. Hopefully the children will thereby improve their ability to

describe and understand their own thoughts.
2) The problem domain to which the AI programs are applied is learned, and in a new and perhaps better way
3) If children are to program, then AI can be an interesting open ended problem domain for that programming
4) The children will learn about AI which is a subject, in the opinion of the author, that is as important as spelling or history.

As an example, consider a child writing a simple natural language understanding system. The child will hopefully learn much about computational linguistics and something about how he or she talks and listens.

To facilitate AI programming of, for example, natural language understanding, or common sense systems, the proper primitives (from the child's point of view) need to be provided. For this purpose the author has developed a set of programs LOGO called "LAIL" for LOGO AI Language. LAIL contains a set of operators useful for writing AI programs. Currently LAIL consists of the following tools:

1) A powerful pattern matcher for natural language understanding
2) A context-free generator for sentence generation to which the child provides rules.
3) A relational data base handler facilitates memory and inference
4) An actor-like animation system which is described later.

Most important of all, these tools should be simple, powerful, natural, and encourage the right kind of conceptual thinking.

AN ACTOR-LIKE ANIMATION SYSTEM

Animation is a programming domain for children in which I am particularly interested. My dissatisfaction with the present animation primitives in LOGO led to the design and implementation of an animation system written in LOGO. Each object in the system is very similar to an "actor" in Carl Hewitt's formalism (Hewitt, 1973). An actor is a computational entity that can receive and transmit messages. Each object in my system can accept turtle-like commands (e.g. forward), remember items told to it, and be taught how to handle new kinds of messages. Each object has a set of patterns which are matched against the incoming message. If a pattern matches then the action associated with that pattern is invoked. Parallel movements of objects on the display screen are handled by an actor called a "scheduler". When the scheduler is sent a message asking it to run, it passes those messages it has associated with the time, redisplays the screen, increments the time, and loops. The scheduler can also produce actors called "movies" that can show a cartoon at any reasonable speed. An example of the use of this system is shown in Figure 3.

The view of programming as collections of actors, or a community of "little people", that send and receive messages from each other is very powerful. It is conducive to a modular, simple, natural representation of the knowledge needed

SQUARE [MAKE GEORGE]

This sends the message "make George" to SQUARE, which creates a square named George

GEORGE [REMEMBER SPEED 25]

This tells George his speed of movement

GEORGE [IF RECEIVE ZIG ZAG ?N THEN ZIG.ZAG "GEORGE :?N]

George is told that if he receives the message "zig zag" followed by some amount, then he is to call the appropriate procedure

CIRCLE [MAKE SALLY]

A circle named Sally is created

SALLY [DO FORWARD 100 AT FRAME 2]

Sally is told to go forward 100 at movie frame 2. This problem is passed on to the scheduler

GEORGE [DO ZIG ZAG 75 AT FRAME 2]

George is told to zig zag at frame 2

SCHEDULER [GO]

The scheduler is told to go. After 2 frames the appropriate messages are sent to Sally and to George, producing the following cartoon:

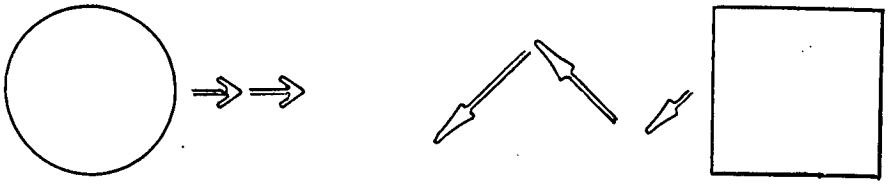


FIG. 3. An ACTOR-like animation system, an example of its use

for the application. The model for intelligence can be a community as easily as it can be an individual with an actor system.

Another AI aspect of this system is the explict "kind-of" hierarchy of actors. Each object is told what class it is a member of when it is created. When any object received a message it cannot handle it passes the problem on to the class of which it is a member. An example of such a tree is shown in Figure 4. The important concepts of instantiation, class membership, placement of knowledge

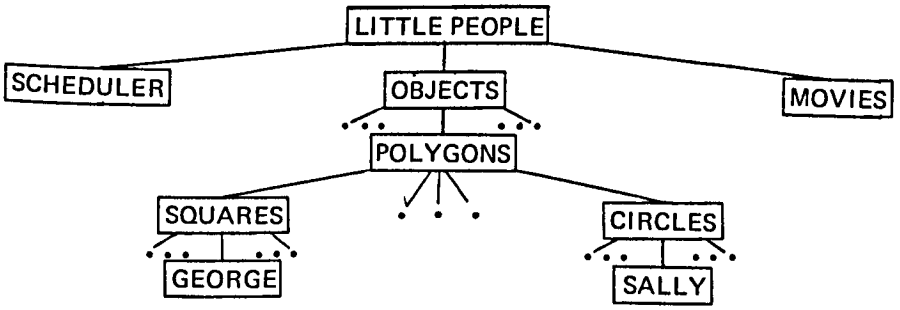


FIG. 4. Actor hierarchy

at the best level of generality, inheritance of properties, and exceptions hopefully flow from the proper use of this aspect of the system.

AN INTELLIGENT ANIMATION SYSTEM

Another environment for learning powerful ideas is one in which the child interacts with specially designed AI programs. I am in the process of designing an intelligent animation system as a test of this hypothesis. In the normal mode of interaction the child will tell a story concerning the "Peanuts" characters, and the system produces an animated cartoon based on the story. The child can then describe the modifications he or she desires. The system produces a new cartoon and the process iterates until the child is satisfied (or quits). The usual concepts of linearization, debugging, and the like prevail here without the need to learn a computer language. Hopefully the child will learn about animation, film making, and story telling by writing stories and scripts to produce the desired cartoons. An hypothetical example of a conversation with this system is shown in Figure 5.

User: Charlie Brown is walking and meets Lucy. He says, "Good morning". She says, "What's so good about it?...". Charlie Brown frowns and says, "Good Grief!".
The system shows the cartoon after having made many simple inferences and default choices. For example, it decides where to put Lucy and Charlie Brown, how fast they walk, how they are oriented, their facial expressions, and so on.
System: How was that?
User: OK, but Lucy should look crabby when she says, "What's so good about...".
The system changes the necessary parts of the cartoon and shows the new one.
User: That's fine

FIG. 5. A hypothetical discussion with the intelligent animation system.

Another more important way in which a child could interact with this planned system is by understanding and modifying the knowledge of the system. The system's knowledge about animation techniques, the real world, common sense facts about people and things, and the "Peanuts" characters will be as modular and simply represented as possible. The child, with the help of the system, will be able to define new characters for the stories, change the system's vocabulary, add a reasonable default course of action and so on.

One hope is that in understanding and modifying the knowledge of an intelligent system the children will actually see the knowledge of something actually accomplish something complex. The idea that knowledge and intelligence can be something formal should help the children to be more explicit about their own thoughts and knowledge. The use of the system will also, hopefully, help provide a vocabulary for talking and thinking about one's thoughts. If the design and implementation of this intelligent animation system is done well, it will provide a rich and exciting supplement to LOGO's traditional environment.

SUMMARY

In this paper three ways in which AI can interact with education have been described. After an introduction to LOGO thinking and language, the benefits of children writing simple AI programs using the proper tools were described. Finally, the ways in which an AI system designed for education can interact with children were discussed. These ideas should be implemented and tested with children. Only then will the effects on education be known.

REFERENCES

Abelson, *et al.*, (1974) Logo Manual, August 1974, LOGO memo No. 7.

Goldstein, I.P. (1974) *Understanding Simple Picture Programs*, MIT AI Laboratory AI-TR-294.

Hewitt, C., *et al.*, (1973) A universal modular ACTOR formalism Proc. *IJCAI-73*, Stanford, California.

Papert, S. (1971) Teaching children thinking, MIT-AI Laboratory Memo No. 247.

Papert, S. (1975) Uses of technology to enhance education, MIT-AI Laboratory Memo No. 298.

Sussman, G.J. (1975) *A Computer Model of Skill Acquisition*, American Elsevier Publishing Company, Amsterdam.

CASE STUDIES IN EMPIRICAL KNOWLEDGE

21

Knowledge Representation for Archaeological Inference

James Doran
Department of Computer Science
University of Essex

INTRODUCTION

In this paper I shall be concerned with the problems of interpretation which archaelogists encounter when they study excavation data from prehistoric cemetery sites, and I shall consider these problems in the light of the evolving artificial intelligence theories of specialist knowledge representation.

Prehistoric archaelogy is a subject dominated by the need to collect and piece together fragmentary and diverse evidence to form a coherent overall picture. In this task archaelogists have for some years now made increasing use of various mathematical models and techniques of statistical data analysis (Doran and Hodson, 1975). But these techniques are greatly limited by their inability to embody the specialist knowledge required in typical archaeological inference contexts. It is therefore natural to ask whether formal techniques of interpretation can be developed which 'know' something about archaeology. As I discuss this question, and the form which a possible interpretive system might take, I shall try to present both the broad archaeological background and sufficient actual archaeological detail for the issues to come to life.

Archaeology has clear attractions as a problem-domain for artificial intelligence research. Many of the problems of recognition and interpretation encountered in archaeology have close parallels with classic artificial intelligence problems, notably those of scene analysis. But archaeological problems have their own characteristics and need special treatment. Further, any development of 'knowledgeable' interpreters of archaeological data seems bound to force an instructive investigation of the logical relationship between such systems and existing archaeological tools such as factor analysis and computer simulation modelling.

PREHISTORIC ARCHAEOLOGY

The fundamental objectives of prehistoric archaeology are to reconstruct and

to explain the evolution of human society up to the time when written records begin to be available. Archaeology is thus one of the social sciences. But compared with 'mainstream' social sciences such as political science or sociology it has two noteworthy characteristics: the enormous time-span that is its domain, and the relatively 'concrete' nature of its evidence. The great time-span covered by archaeology means that it has a potentially crucial contribution to make to social theory, which is often criticised for being too concerned with structure and not enough with long-term dynamics. Archaeological evidence is 'concrete' by comparison with that of, say, sociology since it is typically formed of directly observable objects or traces rather than of statements by or about people. This is clearly both a strength and a weakness. It makes archaeology somewhat more of a hard science (and archaeologists are often aided by physicists, chemists and biologists) but removes it further from its ultimate social goals.

Paradoxically archaeological data are both embarrassingly abundant and typically insufficient. The evidence collected from field surveys and excavations of prehistoric hunting camps, settlements, fortifications, burial mounds and the like, elaborated by specialist studies (carbon-14 dating, pollen analysis, dendrochronology, magnetometer surveys, bone studies) is typically too extensive to be adequately processed, whence current interest in computer-based methods of data capture and handling. But even so it is unusual to be able to work out with any confidence just what was happening and why in some particular locality at some particular time. It may well be possible to decide that over a period of some five hundred years there was a sequence of four successive houses on some spot with such and such individual characteristics. But it is likely to be much more difficult to work out the economics and trading links of the settlement of which the house, in its successive incarnations, was a part. And social organization is probably just a matter for speculation. The source of the difficulty is partly that the evidence available (there may be much still in the ground) is simply insufficient, partly that the complexity of the inference problem becomes too great, and partly the general weakness of archaeological theory.

Strictly, the task of unravelling the past of human society cannot be split into independent subtasks. Every act of archaeological interpretation is influenced by the results of every other act. But, of course, archaeologists simplify: the subject is broken down into specialities by period or region or type of evidence. Analogously I shall now concentrate on the interpretation of prehistoric cemetery sites, while recognising that this is to some extent a dangerous and potentially misleading thing to do.

CEMETERY SITES AND THEIR INTERPRETATION

Societies almost always dispose of their dead with care and elaborate ritual. Burial, and therefore extended survival of remains, is common. It follows that there are two reasons why funerary evidence is particularly important in archaeology: it tends to reflect social organisation, and there is a lot of it. Look in any archaeological museum! To elaborate the former reason a little, not only

is it likely that burial practice reflects the beliefs of a society, but differential treatment of the dead within a cemetery probably closely corresponds to differential status among the living. These assumptions can be misleading but they certainly have general validity. When comparing cemeteries, the regular assumption made by archaeologists is that clearly different burial practices probably correspond to different societies or to the same society at widely different times. Even making only this weak assumption, such evidence has been of enormous importance in archaeological work.

By a (flat) cemetery I mean a collection of relatively simple graves, dug as pits or shafts, without mounds or other surviving surface monument. Even the simplest grave will, however, be capable of extended description. It will, for example, be possible to specify its location, orientation, size and depth. The burial may or may not be multiple, and may be an inhumation or a cremation. If the former, the posture of the skeleton may be recoverable and possibly its age and sex. There will probably be "grave goods"—remains of clothing, jewellery, weapons, pottery vessels and the like. This list is by no means exhaustive. Commonly only part of the data potentially available from a grave will have been recorded. Similarly it is common for only a fraction of a cemetery to have actually been excavated.

It will be helpful to distinguish between the *structural* and the *social* interpretation of a cemetery. By a structural interpretation I mean the process of moving from the descriptions of the individual graves to a description of the cemetery as a whole expressed in terms of appropriate concepts (for example, age and sex distribution, spatial organisation, internal chronology, typical combinations of grave goods). By social interpretation I mean carrying this process of inference further to statements about the society from which the cemetery is derived. Some examples of typical inferences will make this distinction clearer:

> Recognition of coexisting but consistently different burial styles (a structural interpretation) may suggest an established caste structure in the population (social interpretation)
>
> Recurring orientations of the graves in a cemetery (structural interpretation) may prove to be linked to the position of the sun at the time of burial and may even indicate at what seasons of the year an associated settlement was occupied (social interpretation)
>
> A full reconstruction of the growth of a cemetery, including an estimate of its total lifetime (structural interpretation), leads to an estimate of the size and demographic structure of the corresponding population (social interpretation).

An early and mistaken structural interpretation of the famous Hallstatt cemetery (European Early Iron Age) was that it consisted almost entirely of male graves. The equally mistaken social interpretation was that this reflected an all male work force for the nearby salt mines (Kromer, 1959).

Actual archaeological studies of cemetery sites come, of course, from a diversity of geographical areas and chronological periods. Nevertheless the process of interpretation has a certain identity which is independent of any particular cemetery. In particular it is obvious that when interpreting a cemetery, at either structural or social level, the archaeologist brings to the task certain beliefs and assumptions based on knowledge about cemeteries and their relationship to people and societies. Thus he knows or believes that in most populations the sexes are about equally balanced; that cemeteries are laid down over a period of time, not all at once; that it is most unlikely that a women would be buried with a sword; that an empty grave may be the result either of drowning (with the body lost) or of poor excavation technique; that two quite different burial practices can coexist in the same society at the same time. Table I lists some of the more commonly used beliefs and assumptions and will indicate their diversity.

Each belief or assumption allows the archaeologist to recognise certain properties of the cemetery as significant and to draw immediately useful conclusions. Thus the process of sexing graves, typically based on osteological and social assumptions, provides data for the next step in the inference chain. Inference terminates, of course, with conclusions that have no usable implications. To infer from grave orientation that a cemetery is Christian leads nowhere unless we can infer something (other than grave orientation!) from the fact that a cemetery is Christian, for example that it is relatively late in date.

Cemeteries are rarely studied entirely out of context. There is almost always some parallel evidence available though often surprisingly little. For example, there may be similar cemeteries nearby so that it is really a matter of interpreting the cemetery group rather than any one of them. Or a pottery or metalwork typology may be available based on material from dwelling sites, although it is unusual for a dwelling site and its burial ground to be located and excavated together. Naturally archaeologists take such evidence into account whenever it exists. In a fairly obvious sense it forms a task specific extension of the archaeologist's more general knowledge of cemeteries and their properties.

STATISTICAL AND COMPUTER METHODS

The Hallstatt cemetery mentioned above contains well over a thousand graves (excavated, incidentally, as early as the 1850's). Although this number is unusually large, it is common for a cemetery excavation to uncover several hundred graves. Since a grave is itself a complex structure, and may obtain a score or more artifacts each of which is in turn of considerably complexity (for example, an intricately decorated glass bracelet) there is a great deal of data to be considered. The archaeologist is therefore obliged either to be very selective and subjective or to make use of formal methods of analysis.

The latter option has been more and more exercised in recent years (see, for example; Peebles, 1971; Saxe, 1971; Moberg, 1972; Randsborg, 1973; Hodson, 1975). Commonly what has happened is that archaeologists have resorted to

TABLE I

Skeletons are the remnants of people
Age and sex determinations are often unrealiable
A predominance of male or female graves requires explanation
Cemeteries are built up over a period of time
Weapons imply men, spindle whorls imply women
Burial grounds are associated with habitations
Men sometimes die in battle
Design changes, usually continuously and usually for the better.
Very different burial practices can coexist
Deviance is often reflected in burial–an isolated grave, without grave goods, with the skull trepanned, makes sense.
Cemeteries are rarely fully excavated
Some excavators are unreliable
Family structure is often reflected in cemetery organisation
Grave goods may indicate social roles
Specialist craftsmen are rare
People have limited memories–the locations of graves may be forgotten
Artifacts decay at different rates depending upon material and conditions
Metal work may be traded over long distances
A sword might have a lifetime of a hundred or more years
Graves may be oriented by reference to the sun, or to the settlement, or to the "land of the dead"
Exhumation and reburial happens in some rituals.
Secondary burials may be associated with an important primary
The level of the ground surface may change dramatically over millenia
If two graves intersect, they are likely to be well separated in time.
Women bear children and sometimes die in childbirth
A randomly generated set of locations may appear to be patterned
Grave diggers avoid difficult ground
Charcoal indicates a fire
People wear clothes
Adjacent graves are more likely than not to be similar in date.
A young rich woman is likely to wear the latest fashions
Graves are sometimes disturbed or robbed
Most, but not all, people are right handed.

A few of the facts and beliefs commonly employed by archaeologists when they interpret cemetery excavation data. The ordering of the list has no significance. It is an instructive exercise to attempt to recast each of these propositions as one or more "recognition demons" (observing X indicates the existence of a Y with implications Z) of the kind discussed in the text.

relatively straighforward tests of statistical significance at the level of structural interpretation. A typical example is the use of a significance test to check for systematic grave alignment. Often tested too is the null hypothesis that a particular type of artifact is equally common in different sections of a cemetery. Much more elaborate computer based methods of data analysis, notably forms of cluster analysis, of non-metric scaling, and of combinatorial seriation, have also been used. One particular cemetery, that of the la Tène 'culture' at Münsingen-Rain near Berne in Switzerland (Hodson, 1968), has provided the

data for a whole series of experiments and a very brief description of it and what has been done should serve to give the flavour of this kind of work.

The Münsingen cemetery dates from about 500 BC to about 50 BC, and was discovered and excavated in the first decade of this century. It turned out to contain about 200 graves, almost all of them inhumations, of which 77 provided skeletal remains which could be sexed (see Figure 1). Also in the graves was a great deal of sophisticated metalwork in iron, bronze and silver, including notably a hundred or more decorated bronze fibulae (brooches). For examples of the grave goods see Figure 2. The importance of the Münsingen cemetery lies in its unique size (for the late Iron Age) and, more specifically, in the possibility of reconstructing the stylistic evolution of the la Tène metalwork designs by recovering the internal chronology of the cemetery, that is, the order in which the graves were laid down. If the evolution of metalwork design can be established in detail, then it can be used to refine the dating of la Tène material found elsewhere.

An initial guide to the chronology of the cemetery is provided by its roughly linear layout (see Figure 1) together with background knowledge of la Tène metalwork which indicates that the northern end of the cemetery is relatively early and the southern end late. This is the starting point for the most important of the conventional studies of the cemetery, that of Hodson (Hodson, 1968). Hodson developed a full artifact typology for the cemetery (that is, the set of artifacts found in the cemetery is divided into classes and sub-classes ('types') first on functional and then on morphological and stylistic grounds). He then used this typology to arrange the graves into their most likely true chronological sequence ('chronological seriation'), invoking the principle that graves are roughly contemporaneous to the extent that they contain artifacts of the same type.

A very different study is that of Schaaff (Schaaff, 1966) who investigated the spatial organisation of the Münsingen cemetery and of other smaller cemeteries of the same type (notably that at Andelfingen–see below). He detected a tendency for graves of a particular age/sex class (male, female, or child) to be grouped together. And Martin-Kilcher (Martin-Kilcher, 1973) has studied in detail the ways in which customs of personal adornment changed during the lifetime of the Münsingen cemetery. For example, women wore neck torcs in the earliest phase of the cemetery, but never thereafter. She succeeded in distinguishing changes in adornment custom from changes in burial custom and from changes in the structure and styling of jewellery. Martin-Kilcher also suggested that the Münsingen cemetery corresponded to a small farming community of not more than three or four households each of about ten persons.

Each of the foregoing studies features the construction of careful arguments based on a mass of detailed evidence. The Hodson study in particular required much laborious description and comparison of artifacts, followed by an equally laborious search for an optimal artifact classification and grave seriation. And this study was, in fact, the starting point for the numerous computer studies

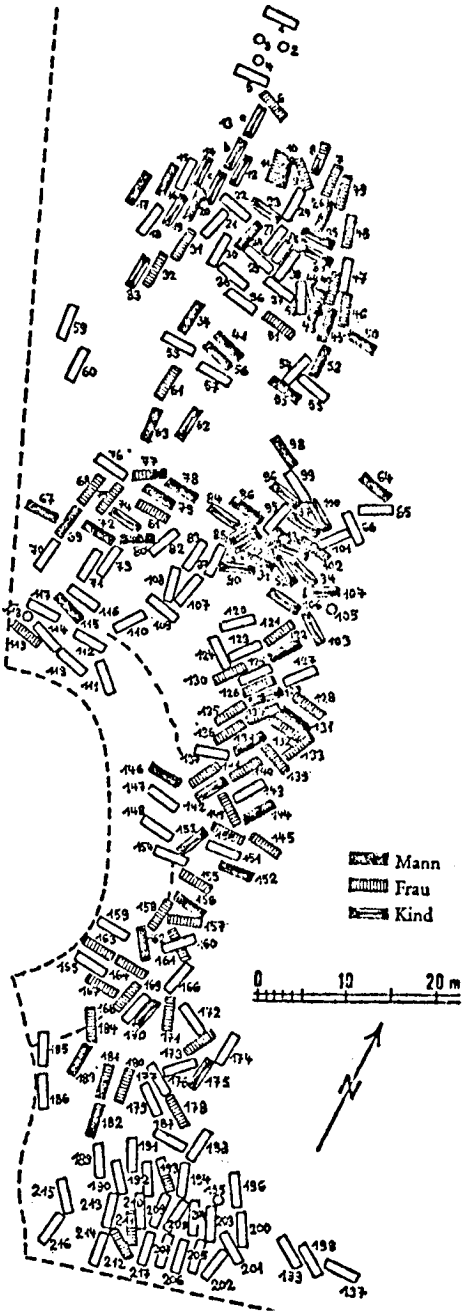


FIG. 1. Plan of the la Tène cemetery at Münsingen-Rain. Graves are specified to be male, female or child wherever determined (Schaaff, 1966, fig. 7).

FIG. 2*i*. A drawing of the skeleton and the objects associated with it

Grave W–E, 190 cms. deep. Stones in half circle by head. Female (?), 40–60 yrs.

Grave goods:
round neck: 373a–b
on breastbone: 369–372
on r. forearm: 367
on r. hand: 368a, 368b (bent ring fragments)
on ankles: 374a–b, 375a–b, in pairs

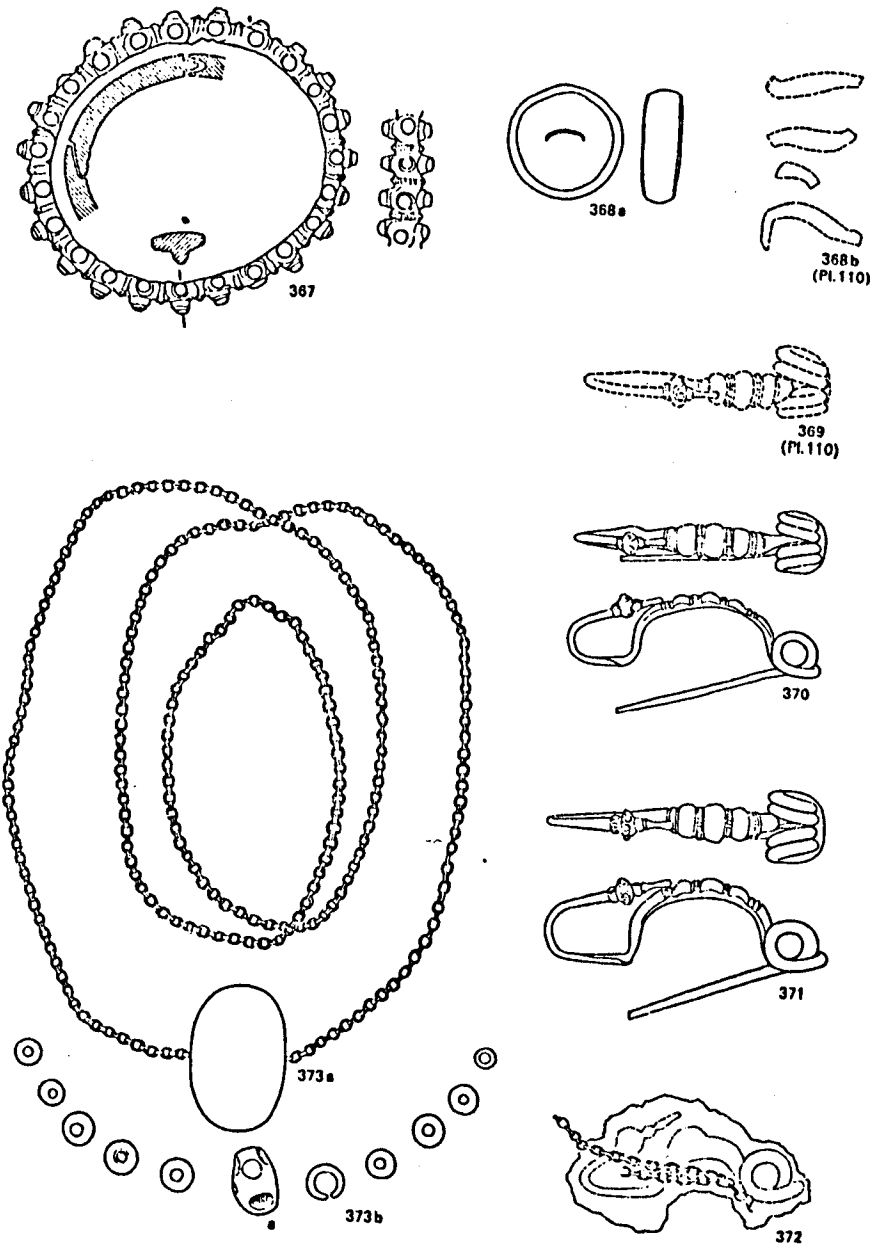
FIG. 2*ii*. The published description of the grave and its contents

FIG. 2. A typical Münsingen grave *[Hodson, 1968, pp57,130]*

which have been made of the Münsingen data. These fall into three broad categories:

(a) cluster analyses (notably K-means analysis) of sets of brooches drawn from the cemetery, with the objective of refining typology
(b) seriation studies of (a subset of) the graves. That ordering of the graves is sought which most consistently puts similar graves close to one another. As noted above, the assumption is that similar graves are chronologically close, so that an ordering derived in this way must reflect the true chronological ordering.
(c) non-metric scaling studies of both the fibulae and the graves with objectives similar to those specified in (a) and (b).

Details of, or references to, virtually all of this work will be found in (Doran and Hodson, 1975). It is worth noting that each of these methods of multivariate



368a-b silver, 372 iron, 373a bronze and amber, 373b (a) amber and blue glass, the rest bronze. Scale 3:4.

FIG. 2*iii*. A drawing of some of the artifacts found in the grave

FIG. 2(cont'd). A typical Münsingen grave *[Hodson, 1968, pp57,130]*

data analysis involves a discrete function optimisation problem which is solved by iterative search. The search techniques involved are simple forms of those explored by artificial intelligence scientists within what Nilsson has called the "state-space" paradigm (Nilsson, 1971). There is also a sense in which each of these techniques *interprets* the data. This aspect is clearest in the case of the seriation studies where the techniques can reasonably be said to propose a particular chronological sequence for the graves.

SOLCEM – A CEMETERY INTERPRETATION PROGRAM

The statistical techniques mentioned in the preceding section, from tests of significance to complex clustering or seriation procedures, have in common that they are applicable only to small and artificially isolated parts of the total problem of inferring from a cemetery excavation record to plausible structural and social conclusions. Typically, their use implies that interactions between different categories of data and between different categories of conclusions are ignored. And this is substantially true of conventional archaeological studies. For example, the Hodson and Schaaff studies of the Münsingen cemetery sketched above treat internal chronology and spatial organisation respectively. But each of these two aspects of the cemetery's structure is unravelled largely as if it were independent of the other, which is manifestly not the case. Similarly, Martin-Kilcher's study of personal adornment at Münsingen accepts the Hodson chronology as a fixed framework, although logically each has implications for the other. Obviously the assumption that interpretive subgoals are independent and their conclusions additive is a highly convenient one. But equally obviously it can be highly misleading.

A few years ago I wrote, in Algol-60, a program called SOLCEM (Doran and Hodson, 1975, pp. 309-315). My purpose was to try to gain a better understanding of just what is involved in the cemetery interpretation problem. SOLCEM is capable of considering the bulk of the evidence in a (small) cemetery excavation record, and of generating an integrated overall interpretation.

Table II lists the main features of the SOLCEM program and Figure 3 is an outline flowchart. The program possesses a range of data analytic capabilities, emobided as a set of *data analysers*. Each data analyser is, roughly, a specialist procedure which may be selected and applied by an executive to a partially developed cemetery interpretation (initially the 'null' interpretation). A data analyser is applicable only when the categories of data and prior results which it requires for its action (typically including results returned by other analysers) are available, and when applied it adds its own conclusions to the specific interpretation of the cemetery being developed.

Since data analysers can and often will return sets of alternative conclusions, and since there is usually more than one analyser applicable to any particular partial interpretation, it is sometimes appropriate for SOLCEM to explore alternative interpretations in parallel and it typically does so. The program's executive chooses which partial interpretation of those currently being developed to

TABLE II

The cemetery data used by SOLCEM comprises:

- any prior artifact classification
- artifact descriptions
- artifact-grave associations
- grave locations
- age/sex determinations for graves
- any established chronological relationships between pairs of graves

The capabilities of the eight data analysers are:

(i) classify artifacts from their descriptions
(ii) detect associations between artifact classes and age/sex grave categories
(iii) infer age/sex of indeterminate graves from artifact class associations
(iv) analyse spatial organisation of cemetery
(v) infer age/sex of indetermine graves from spatial organisation
(vi) infer chronological sequence of graves from their contents
(vii) integrate findings of analyser (iv) with those of analyser (vi)
(viii) reconstruct the evolution of the artifact types during the lifetime of the cemetery.

A complete SOLCEM cemetery interpretation involves:

- the chronological sequence of the graves
- the spatial organisation of the cemetery
- age/sex determinations for previously indeterminate graves
- patterns in the assignment of grave-goods to graves
- the evolution of the artifact types in antiquity

A summary of the capabilities of the SOLCEM cemetery interpretation program. The program generates an integrated interpretation of the cemetery taking into account the interactions between different types of evidence. But in any particular respect, for example the analysis of spatial organisation, its capabilities are very limited.

pursue next by reference to estimates of plausibility and internal consistency calculated by the data analysers themselves. The choice of which analyser actually to apply to the chosen interpretation, from those currently applicable, is made by reference to a simple priority list.

In this way the different analytic capabilities of SOLCEM are interwoven. Effectively the program searches through the space of partial interpretations of the given cemetery data looking for a complete interpretation which is sufficiently plausible to be finally accepted. In form a complete interpretation is a structural description of the cemetery's generation, covering the order of deposition of the graves, an estimate of the rules by which they were located and grave goods assigned to them, and an estimate of artifact type evolution. It roughly corresponds to a stochastic simulation model embodied in an auxiliary program, SIMCEM, used in experimentation to simulate the generation of cemeteries and hence to provide test data for SOLCEM.

Although SOLCEM copes satisfactorily with cemetery test data extending to

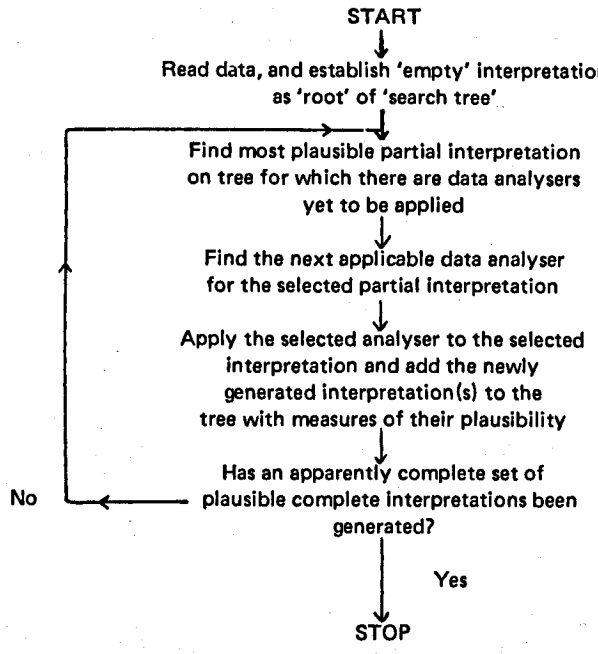

START
Read data, and establish 'empty' interpretation as 'root' of 'search tree'
Find most plausible partial interpretation on tree for which there are data analysers yet to be applied
Find the next applicable data analyser for the selected partial interpretation
Apply the selected analyser to the selected interpretation and add the newly generated interpretation(s) to the tree with measures of their plausibility
Has an apparently complete set of plausible complete interpretations been generated?
No
Yes
STOP

FIG. 3. Outline flowchart of the SOLCEM program.

about a dozen graves, it has never been applied to any significant body of real cemetery data. This is because it is impossible to justify the substantial effort needed to prepare an actual data set sufficiently large and detailed to be realistic, when it is clear that the program in its existing form would not produce sensible answers. Experience with SOLCEM has demonstrated that its performance is limited by the following two facts:

(i) for any realistically sized cemetery, the total inference problem is computationally extremely demanding (which is, of course, why archaeologists do not attempt it, but always simplify), and
(ii) alongside statistical techniques the problem requires the application of substantial specialist and general knowledge.

There are thus two directions in which improvements to SOLCEM might be sought. The problem of computational overload might be attacked by seeking an efficient solution strategy involving, no doubt, a trade-off between accuracy and effort. This would certainly raise deep issues of statistical estimation and general problem-solving theory. Secondly, an attempt might be made to render SOLCEM much more knowledgeable in a general sense. This is the possibility that I shall pursue here.

The point is that although SOLCEM is statistically quite intricate, it has all too little actual knowledge of cemeteries (see Table I) and would certainly be

blind to important aspects of real cemetery data. Worse, such knowledge as it has is irregularly scattered through the data analysers and the executive. On the other hand, efficient utilisation of knowledge should not only eliminate SOLCEM's "blind spots" but also go far to control the combinatorial explosion.

Initially one might suggest that there should be not eight, but eighty or even eight hundred data analysers. But this is a recipe for computational collapse unless coherent principles of organisation and knowledge representation are followed, and it is far from clear what these principles should be. It is natural to look for guidance to current artificial intelligence work on knowledge representation for recognition and inference.

ARTIFICIAL INTELLIGENCE TECHNIQUES FOR KNOWLEDGE REPRESENTATION

A variety of knowledge representation schemes have been explored in artificial intelligence work including predicate logic assertions, semantic networks, procedures, production systems, and frames. But there exits no hard theory comparing these representation schemes and capable of indicating which will be the most useful in any particular application. So all that we can do here is to look at the most prominent and recent of the schemes which have been proposed, and to try to extract concepts and methods which seem likely to be useful in designing a successor to SOLCEM.

Production systems have been used successfully in a number of artificial intelligence applications projects, notably in the Heuristic DENDRAL project (Buchanan and Lederberg, 1971) in analytic organic chemistry, and in the MYCIN project (Davis, Buchanan, Shortliffe, 1975) in medical diagnosis. Further, the basic idea of a production is that it is a rule of the form "if A is encountered, then do X." In a general way, therefore, a system or list of productions already bears at least a passing resemblance to the SOLCEM set of data analysers. But this promising line of thought founders on the observation that all the SOLCEM data analysers are vastly more elaborate than the productions in any implemented production system. Short of breaking down such archaeologically fundamental activities as artifact typology or chronological seriation into sets of productions, which even if possible would hardly be sensible, production systems look too simple to be useful.

A working artificial intelligence system which is a major and effective elaboration of the production system philosophy is the HEARSAY II speech understanding system (Erman and Lesser, 1975). Committed to the 'hypothesize-and-test' view of inference, HEARSAY II is structured as a number of independent 'knowledge sources' each of which can act upon a uniform, multilevel, hypothesis structure, the 'blackboard', which *is* the system's understanding of the speech data put to it.

Each knowledge source is a kind of 'super-production' which, if and when its preconditions are satisfied in the blackboard, will be activated and will create, modify or delete hypotheses as appropriate in context. It is fundamental to the

philosophy of HEARSAY-II that knowledge sources should be quite independent, should communicate only through the blackboard and should in no sense 'call' one another with the rigidity of control that would imply. Predictably this stand leads to problems in achieving efficient and goal-directed behaviour, and HEARSAY-II has additional and not entirely elegant mechanisms directed to overcoming these problems.

The currently most debated proposal for knowledge representation is the suggestion that knowledge should be structured as 'frames'. The debate was initiated by Minsky (Minsky, 1975) who suggested that:

> "Whenever one encounters a new situation..... he selects from memory a structure called a *frame*: a remembered framework to be adapted to fit reality by changing details as necessary.
>
> A frame is a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party"

Minsky conceived of frame-selection as an essentially "top-down" process and of frames themselves as having a great deal of procedural knowledge attached to them concerned with, for example, what should be done if a selected frame turns out to be inappropriate.

Minsky's ideas have been developed and elaborated in recognition contexts by, among others, Fahlman (Fahlmann, 1973) and Kuipers (Kuipers, 1975). Bobrow and Winograd (Bobrow and Winograd, 1975) are implementing a new programming language (Knowledge Representation Language – KRL) which is largely based on frames ideas and which should therefore facilitate the development of functioning frames based systems. But in spite of this activity there is as yet nothing very concrete which might be adopted as the basis for a successor to SOLCEM. Currently frames ideas provide no more than useful guidelines.

Of these three approaches to knowledge representation–production systems, the HEARSAY II organisation, and frames–it is the HEARSAY II organisation which looks most useful from the archaeological standpoint and which I shall therefore pursue. But before I do so it is worth noting that all three approaches represent knowledge as a set of independent procedural or semi-procedural units. This facilitates identification and modification of the knowledge base. In both HEARSAY-II and frames systems the pattern of activation or utilisation of the knowledge units is fully data-driven, in contrast to the partially predetermined activation pattern of production systems and, indeed, of SOLCEM.

SOLCEM D – A PROJECTED SUCCESSOR TO SOLCEM

Let SOLCEM-D be the name of the proposed successor to SOLCEM. Following the HEARSAY-II system, I shall suppose that SOLCEM-D is structured as a set of knowledge bearing units, the successors to the SOLCEM data analysers, which independently act upon an evolving interpretation structure. However I

shall refer to the knowledge bearing units in SOLCEM-D as *recognition demons* rather than 'knowledge sources' since they have something in common with the demons proposed by Selfridge and Charniak. Further they correspond more to specific concepts (interpreting the word "concept" in a very broad sense), than to broad sources of knowledge. Admittedly this distinction is far from precise. I shall refer to the evolving interpretation structure simply as the *data-base*. The task of each recognition demon is continuously to scan the data base looking for examples of the concept which it embodies. When a demon discovers an example of its concept (in *its* judgment) then it generates and adds to the data-base an *instance*. An instance is, essentially, a description of the specific properties of the concept example found. For example, a demon might specialise in recognising isolated groups of graves. Then each instance that it generated would describe a particular grave group and its immediate properties, such as its location and the number of graves comprising it. Since instances once generated may be inspected and utilised by any other demon, the consequences of a single act of recognition can be propagated throughout the growing interpretation structure.

This is no more than a first sketch for SOLCEM-D. Any attempt to fill in details soon encounters difficulties familiar from the frames debate and from the HEARSAY II work. To see just what form these difficulties take in the context of cemetery interpretation we can usefully look at more specific examples.

ANDELFINGEN AND NEBRINGEN

The la Tène cemeteries at Andelfingen in Switzerland (Viollier, 1912) and at Nebringen in South Germany (Krämer, 1964) are smaller and simpler than that at Münsingen, but they are of the same general period and type. Figure 4 is a sketch plan of Andelfingen showing the locations of the graves together with their orientations and sex (male, female, child) where determined. It is clear by inspection that:

- the graves are almost uniformly aligned N-S or NNE-SSW
- the graves fall into two spatially separated groups, labelled A and B on the plan
- it is possible to divide the graves into four clusters (F1, F2, M and C on the plan) so that each cluster is made up entirely of graves of one sex (with one child's grave somewhat anomalously placed)

Figure 5 is a similar plan of the Nebringen cemetery. Here it is clear that

- the prevailing grave orientation is again NNE-SSW
- the graves fall into five spatially distinct groups (A,B,C,D,E).

These plans are somewhat misleading because they present the sexes of the graves as given. In fact sex determinations were made both from skeletal remains and from a study of the grave goods associated with the burials. Where these two

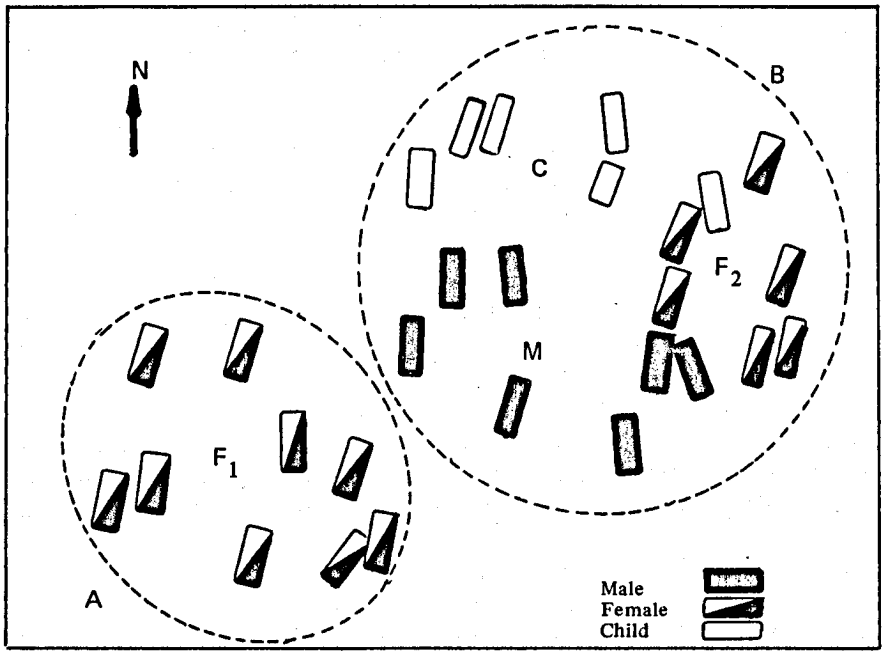


FIG. 4. Sketch plan of the Andelfingen cemetery indicating age/sex determinations and the grave groupings mentioned in the text. (redrawn from Schaaff, 1966, fig. 1).

sources of evidence contradicted one another, then a choice had to be made between them. This choice might reasonably have been influenced by the perceived overall structure of the cemetery, but neither of the authors of the studies concerned admit to having reasoned 'top-down' in this way.

At Nebringen the spatial groups were taken by Krämer to reflect in some way the family structure of the population. Each group contains at least one adult male and one adult female and no group contains more than three adults *provided* that spatial group A is treated as two adjacent family groups (see Figure 5). This is again an example of high-level organisation, once recognised, influencing the interpretation of lower-level detail.

At neither cemetery has it seemed reasonable to try to work out the chronological sequence of the graves. Nor have any significant social conclusions been drawn. For example, no interpretation has been offered for the prevailing NNE-SSW grave alignment. However Schaaff (Schaaff, 1966) has stressed the potential significance of the contrast between the organisation of the Andelfingen and the Nebringen cemeteries. He has suggested that the existence of two different burial customs among the Celts in the Early Iron Age may help in the determination of their later migration routes.

What we see, even from cemeteries as simple as those at Andelfingen and Nebringen, is that SOLCEM-D must be able not only to establish relevant structural features of a cemetery from imperfect evidence, but must also be able

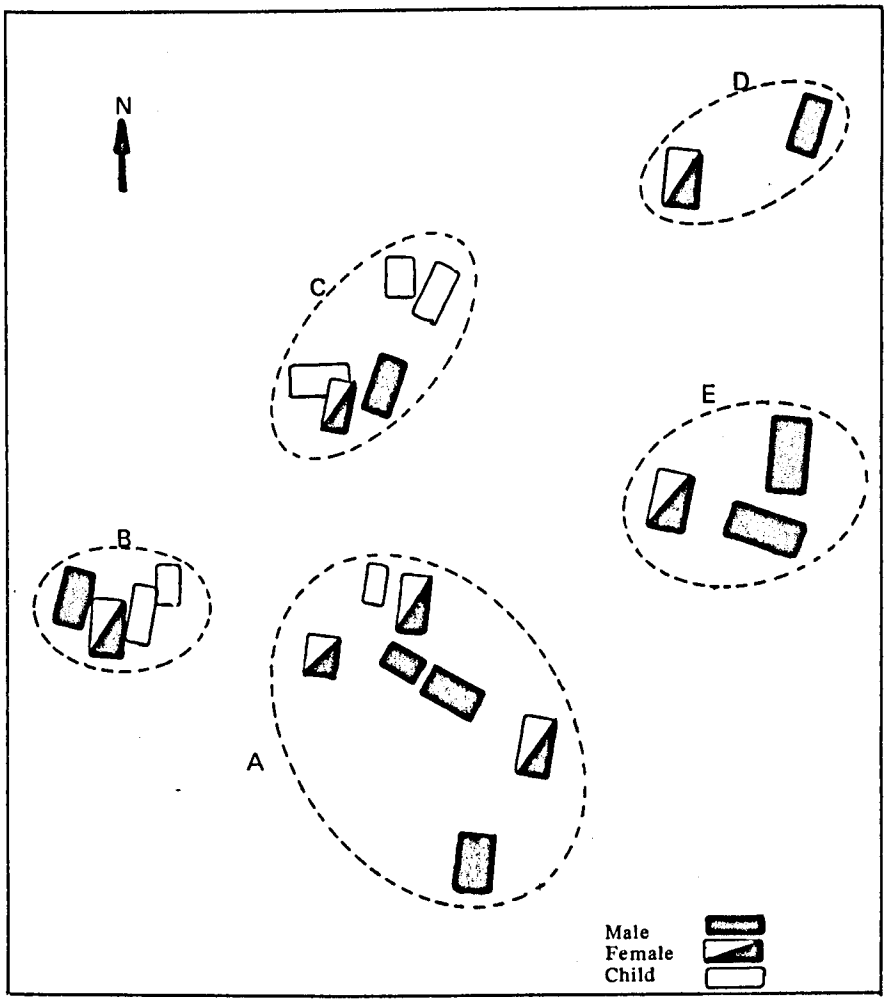


FIG. 5. Sketch plan of the Nebringen cemetery indicating age/sex determinations and the grave groupings mentioned in the text. (redrawn from Krämer, 1964, plan 2).

to reason "top-down" where appropriate. More generally, it must clearly be able to compare alternative interpretations opting for that with the greater internal consistency and plausibility. Notice that to a strictly limited extent each of these capabilities is already possessed by the existing SOLCEM program.

ISSUES IN THE DESIGN OF SOLCEM-D

Imagine now that SOLCEM-D, as sketched earlier, is equipped with a repertoire of recognition demons variously able to recognise spatially isolated groups of graves, to recognise that a particular grave group has a prevailing alignment, to recognise that a set of grave groups all have a common structure itself recog-

nisably a 'family' structure, and so on. Assuming that the data base is initialised with, say, the details of the Andelfingen cemetery (a set of grave instances and perhaps a cemetery instance) then it is plausible enough that the demons would in due course generate instances corresponding to the features of the cemetery noted earlier. But between this observation and an effective working system there stand (at least) four major and interrelated clusters of problems.

Firstly there is the problem of *mapping knowledge into demons*. By this I mean that the recognition and inference capabilities which the system is to have must somehow be allocated to a finite set of demons. Each demon must be able to 'see' its concept given only imperfect and fragmentary evidence. Each must compute the 'right' set or properties for the instances it generates.

The precise range of demons available in the system might seem immaterial provided that all the knowledge needed is embodied somewhere. On the other hand, it is a commplace that using the right concepts can be crucial to understanding. The internal structure of demons must surely include multiple representations of the ideal concepts which they embody.

Related to the foregoing problem is that of how best to handle *instance properties*. As demons generate instances they associate with them their immediate properties. Exactly what form do these properties take, and what is the difference between a property and a concept with its own demon? Are relations properties or concepts or both? Where instances are hierarchially organised how much redundancy is permitted? For example, a common grave alignment throughout a cemetery implies that the same common alignment exists in any subset of the graves. How should such properties be handled?

The problem of *control* is fundamental. The idea that all the demons can be continuously active is computationally quite unrealistic. How, therefore, should the system be structured so that at any time the "right" demons are active and are investigating the "right" structures in the data base. Working top-down is a special case of this: it implies that a high-level demon has activated demons to look for the evidence it requires. Another facet of the control problem is that the generation of a new instance may well imply the recomputation of part, but not all, of one or more existing instances. How can this be handled efficiently?

A possible general strategy is to enable one demon to activate another direct, passing relatively complex 'instructions'. This seems to require that each demon must be knowledgeable about a range of other demons and their capabilities, and so runs counter to the idea that demons should be independent of one another. An alternative, essentially the PLANNER philosophy of invocation by pattern matching, is to enable demons to specify objectives without specifying which demons are to achieve them. Either way, control may or may not be automatically returned to the initiating demon. A further possibility, reminiscent of human thought processes, is that a quite simple network of "relevancy" links might be improved upon both the demons and their instances (possibly utilizing links already existing for other purposes) which would determine patterns of activation of a relatively unstructured but nonetheless effective variety.

Finally there is the problem of creating, evaluating and choosing between *alternative interpretation structures*. The problem ranges from deciding whether or not a particular grave should be included in a recognised grave group, to elaborating and then choosing between radically different complete cemetery interpretations each of which is consistent and reasonably plausible.

It is natural to suggest that each instance generated must have its own credibility rating. It also seems plausible that some explicit mechanism or inherent tendency must be devised whereby elaborate interpretation structures (which in this context might be better called "hypothesis structures") persist whilst fragmentary structures are automatically discarded. One way of achieving such an effect might be by adding to the system's basic repertoire of demons others which recognise not patterns of data but patterns of inference. Human beings have, for example, the concept of a logical implication or of a *reductio ad absurdum* argument and are able to reason *about* particular arguments. A similar capability in SOLCEM-D might be the right way to introduce more sophisticated reasoning processes.

Few of the problems and suggestions sketched above are particularly new. Most have been considered in some detail either in the context of the knowledge representation schemes cited earlier or in connection with new A.I. programming languages such as PLASMA and KRL. Unfortunately only piecemeal progress has been made, and anything like a comprehensive understanding of all these issues still seems very far off.

KNOWLEDGE REPRESENTATION, DATA ANALYSIS, AND SIMULATION

Although sufficiently complex to motivate discussion of current artificial intelligence research issues, the Andelfingen and Nebringen cemeteries are relatively simple by archaeological standards. An archaeologist would not feel the need of statistical or computer aid when studying them, as he might well do when confronted with cemeteries of the size of Hallstatt or even Münsingen. This raises the question, mentioned in the introduction to this paper, of the exact relationship between the ideas of knowledge representation and use which artificial intelligence offers to archaeologists, and the ideas behind the statistical and computing techniques which archaeologists have already begun to use. These apparently very different, even hostile, clusters of ideas must somehow be brought into logically coherent relationship.

Social scientists generally, and archaeologists in particular, typically look to multivariate data analysis (cluster analysis, factor analysis, non-metric scaling) when they are confronted with a great deal of relatively homogeneous data which exhibits no immediately recognisable 'structure.' The recourse to multivariate analysis is partly an attempt to *uncover* meaningful structure, partly an attempt to *impose* structure, meaningful or not, in order to render the data tractable.

If this is an accurate characterisation of the use of multivariate analysis in archaeological work, then it seems that particular techniques can and should be

embodied as individual demons in the hypothetical SOLCEM-D system, but that they will make rather unusual demons–demons which are computationally very demanding (typically involving major iterative searches), which have a very wide range of application within the data base and which generate correspondingly non-specific instances. How important a role such demons would play in typical problems of archaeological inference is difficult to say. The use of multivariate techniques in archaeology is still highly experimental. Everything suggests that they are a kind of 'last resort' for use when knowledgeable interpretation of data is not possible.

The use by archaeologists of computer simulation as an investigatory tool is an even more recent development than their use of multivariate analysis. While discussing the SOLCEM program I mentioned my own use of a kind of stochastic simulation model to generate artificial but realistic cemetery excavation records. But much more significant than this is the use by American archaeologists of such models to infer from field and excavation data to social processes in antiquity (Doran and Hodson, 1975, chapter 11). The growing use of computer simulation is a methodological reflection of the current emphasis on process in prehistoric archaeology, with a corresponding rejection of the traditional, if usually unstated, view of the past as a series of static situations.

It is clear that SOLCEM-D (and SOLCEM for that matter) *can* construct an interpretation which embraces the time dimension. This was implicit in the earlier suggestion that a chronological seriation algorithm might be embodied in a recognition demon. As the interpretation is built, time relationships are built into it. But interpreting a particular body of data by fitting a computer simulation model to it is a little different. It corresponds to the validation stage in a standard simulation study. The simulation is run repeatedly, with successive modifications and varying parameter settings, until a convincing match to the data is achieved. This 'tuned' model then *is* the interpretation sought.

A simulation model is clearly itself a repository of knowledge, but in a much more structured and rigid representation than any of those mentioned earlier. What is the relationship between a computer simulation study and the kind of analysis which SOLCEM-D would perform? It is tempting merely to suggest that a simulation model should be assigned to a single demon as was proposed earlier for statistical techniques. But this is surely absurd. Rather, the event and process generators out of which the simulation model is constructed must individually be represented as demons. But if this is so, how are the demons concerned to be persuaded to cooperate within the loose control structure proposed for SOLCEM-D? There is no obvious answer, although it is not too implausible that a crude form of dynamic modelling might emerge as the temporal aspect of the requirement that newly generated constraints must be compatible with those already existing.

Dynamic modelling is of deeper archaeological significance than may at first appear. It is a commonplace to assert the weakness of archaeological theory in particular and of social theory in general. It is very likely that this weakness has

much to do with our inability to develop sufficiently complex process models and to infer from observations to particular instances of them. The drive towards a "systems approach" throughout the social sciences is an attempt to overcome this fundamental weakness, and is as prominent in the interpretation of prehistoric cemetery excavation data as anywhere else [see, for example, (Binford, 1971) and (Rathje, 1973)]. Thus if real insight can be achieved into the relationship between dynamic modelling and knowledgeable reasoning systems, then the impact on archaeology is likely to be as much at the conceptual and theoretical level as at the level of practical interpretation.

REFERENCES

Binford, L.R. (1971) Mortuary practices: their study and their potential. *Memoirs of the Society for American Archaeology, No. 25* pp. 6-29. (*American Antiquity* 36, 3 part 2, July 1971).

Bobrow, D. and Winograd, T. (1975) Unpublished working papers.

Buchanan, B.G. and Lederberg, J. (1971) The Heuristic DENDRAL program for explaining empirical data. *Information Processing 71* (proceedings of the IFIP Congress 1971; ed. C.V. Freiman) 1, pp. 179-88, Amsterdam: North Holland.

Davis, R., Buchanan, B., Shortliffe, E. (1975) Production rules as a representation for a knowledge-based consultation program. *AIM 266*, Department of Computer Science, Stanford University.

Doran, J.E. and Hodson, F.R. (1975) *Mathematics and Computers in Archaeology.* Edinburgh University Press.

Erman, L.D., and Lesser, V.R. (1975) A multi-level organisation for problem solving using many, diverse cooperating sources of knowledge. *Internal research report*, Department of Computer Science, Carnegie-Mellon University.

Fahlman, S. (1973) A hypothesis-frame system for recognition problems. *Working Paper 57, A.I. Laboratory, M.I.T.*

Hodson, F.R. (1968) *The La Tène Cemetery at Münsingen-Rain.* Berne: Stämpfli.

Hodson, F.R. (1975) *Hallstatt Inaugural lecture in the Institute of Archaeology, University of London.*

Krämer, W. (1964) *Das keltische Gräberfeld von Nebringen (Kreis Böblingen)* (Reihe A, Heft 8, Veröffentlichungen des staatlichen Amtes für Denkmalpflege Stuttgart) Verlag Silberburg, Stuttbart.

Kromer, K.(1959) *Das Gräberfeld von Hallstatt* Florence: Sansoni.

Kuipers, B.J. (1975) A frame for frames: representing knowledge for recognition. *Representation and Understanding: Studies in Cognition Science* (eds. D. Bobrow and A. Collins) Academic Press.

Martin-Kilcher, S. (1973) Zur Tracht-und Beigabensitte im keltischen Gräberfeld von Münsingen-Rain (Kt. Bern) *Anseiger für Schweizeriche Archäologie und Kunstgeschichte.*

Minsky, M. (1975) A framework for representing knowledge. *The Psychology of Computer Vision* (ed. P.H. Winston) McGraw-Hill.

Moberg, C.A. (1972) Questions sur l'outillage mathématique d'une archéologie sociologique. *Les Methodes Mathematiques de l'Archéologie* C.A.D.A. – C.N.R.S. Marseille, pp. 230-248.

Nilsson, N.J. (1971) *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill.

Peebles, C.S. (1971) Moundville and surrounding sites: some structural considerations of mortuary practices II. *Memoirs of the Society for America Archeology*, No. 25, pp. 68-91. (*American Antiquity* 36, 3 part 2, July 1971).

Randsborg (1973) Wealth and social structure as reflected in bronze age burials–a quantitative approach. *The Explanation of Culture Change* (ed. C. Renfrew), pp. 565-570, London: Duckworth.

Rathje, W.L. (1973) Models for mobile Maya: a variety of constraints. *The Explanation of Culture Change.* (ed. C. Renfrew), pp. 731-757, London: Duckworth.

Saxe, A.A. (1971) Social dimensions of mortuary practices in a mesolithic population from Wadi Halfa, Sudan *Memoirs of the Society for American Archaeology*, No. 25, pp. 39-57, (*American Antiquity*, 36, 3 part 2, July 1971).

Schaaff, V. (1966) Zur Belegung latènezeitlicher Friedhofe der Schweiz. *Jahrbuch des Römisch-Germanischen Zentralmuseums, Mainz*, 13, pp. 49-59.

Viollier, D. (1912) Andelfingen, Kt. Zürich *Anz. Schweiz, Altkde.* NF14.

22

Chromosome Classification and Segmentation as Exercises in Knowing What to Expect

Denis Rutovitz

MRC Clinical and Population Cytogenetics Unit
Western General Hospital, Edinburgh

INTRODUCTION

Chromosome analysis is a classic example of context-conditioned pattern recognition. So strong was the conditioning indeed that it having been set down that there were 48 chromosomes in the human cell, all experiments between 1910 and 1946 unfailingly reported 48, at which time culture and slide deposition techniques improved to the point where even the most Pavlovian of observers could not but count 46 (Tjio and Levan, 1965). Those trained in cytogenetics know not only how many chromosomes to expect, but also anticipate their comprising about 10 recognisably distinct groups, and know how many there should be in each of these groups. Nowadays also it is possible by varying the conditions of culture and staining to produce distinctive series of black and white bands running transversely across the chromosomes, which enable individual pairs to be recognised (Paris conference, 1971). However, interest still attaches to the conventional technique, because it is not easy to obtain consistently good banding and the analysis is more time consuming. In addition, both simulation studies and experience indicate that few chromosomal rearrangements which are detectable with banding methods will not be associated with a perturbation of the karyotype detectable by conventional staining (Buckton). We revert therefore to the question of classification by conventional staining (Fig. 1). While other indications, such as "fuzziness" or the presence of satellites–that is small antennae-like projections at the end of some arms of small chromosomes–play some part in classification, it rests essentially on the possibility of subdivision of the 2-dimensional feature space shown in Fig. 2. The variables are size (which may be length, area or integrated optical density) and centrometric index: that is ratio of short arm to whole chromosome size. Since actual sizes of chromosomes vary greatly with the point in the cell division cycle at which fixation occurred, we deal with normalised size–that is, individual chromosome size is expressed as a proportion of the sum for the cell.

However, as can be seen from Fig. 2, this subdivision of the size/centromeric

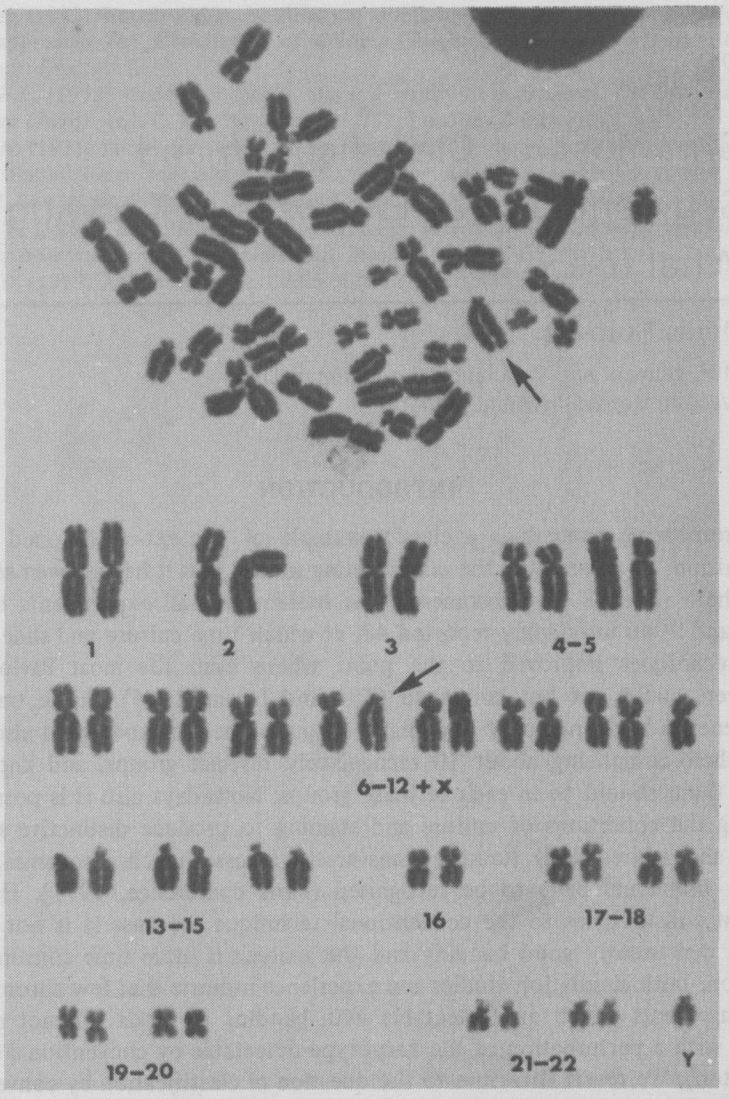


FIG. 1. An orcein-stained cultured human lymphocyte in metaphase, with chromosomes arranged in the 10 groups conventionally recognised by cytologists. The arrow points to a chromosome with an unexpected size/centromeric index configuration–an 'abnormal'.

index plane is at certain points ambiguous. The extent of the ambiguity will depend on the variability of the measuring device and also on the intrinsic biological variability of the chromosomes themselves. Nevertheless humans effect the classification perfectly well. The method followed is usually to start by picking out a distinctive group such as the long acrocentrics. These locate other groups–for example the short acrocentrics may be selected, then the short, medium, and long metacentrics which come in between them and so

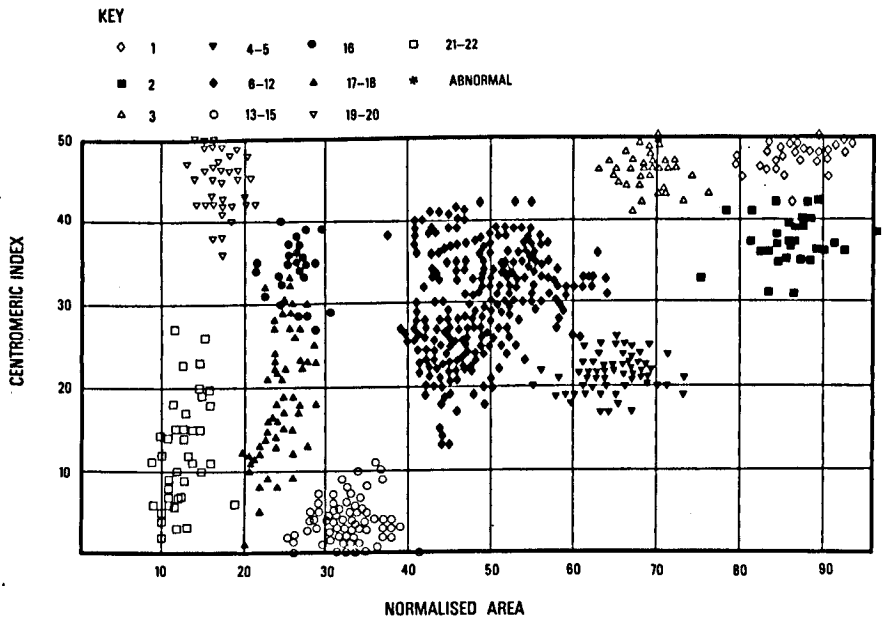


FIG. 2. The size-centromeric index plane used for chromosome classification.

forth. We believe that the procedure depends for its success on classifying absolutely those chromosomes for which the classification is unambiguous, and simply picking the best candidate (or even the first possible noticed) in the ambiguous areas. Indeed it is notable that a human observer cannot satisfactorily classify chromosomes one at a time and requires to work with the entire cell.

Attempts at automatic classification on a purely Bayesian scheme do not succeed, but if full use is made of foreknowledge of the expected karyotype in the way that the human observer seemingly does, reasonable classification efficiency can be achieved. A strategy which attempts to do this is described below.

It is not surprising that knowledge of the normal karyotype should be useful in chromosome classification. It is perhaps less obvious that part of this same information can be used to optimise field segmentation. Almost every image processing undertaking involves field segmentation of one kind or another. Mostly it amounts to setting global or local density thresholds, though sometimes a texture threshold is employed: more rarely outlines are first generated, and regions delineated thereafter. Outline generation is not usually satisfactory in chromosome analysis, due to the presence of both faint and dark regions often with considerable gradients within chromosomes so most workers have concentrated on discrimination by optical density level selection.

An efficient solution to the problem is crucial to successful karyotyping. If the threshold is set too low chromosomes will coalesce into large fractions of the scan field causing problems of two kinds. Since for speed of operation frequent

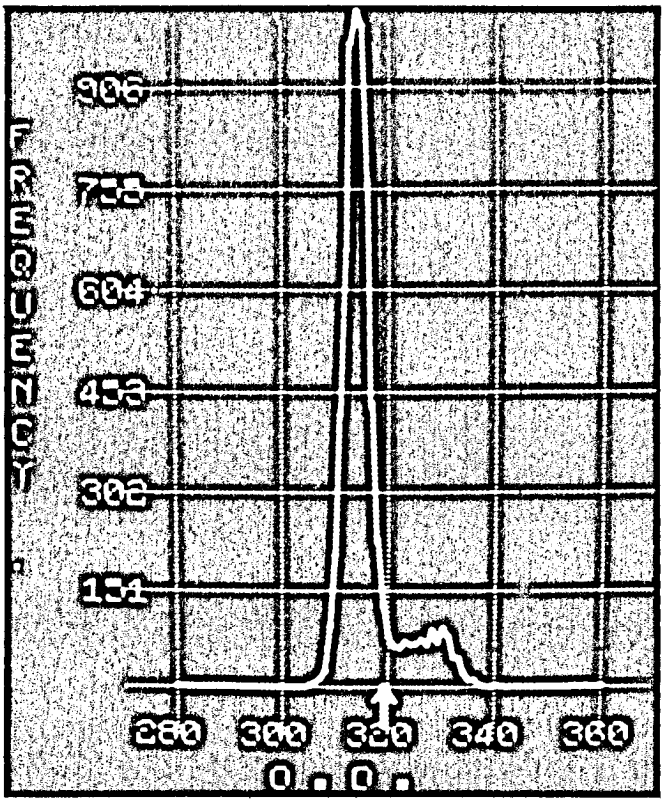


FIG. 3. (a) Histogram of the o.d. levels in the region of the metaphase cell delineated by the selector ring shown in (b).

disc access must be avoided, it is usually assumed that individual field constituents can be accommodated within the main memory of a small computer: the occurrence of abnormally large components will often mean that the cell has to be abandoned. Secondly, for purposes of data reduction and local thresholding it is convenient to use a two pass scan procedure, the first being at a lower resolution than the second. During the fine scan components drawn from the first, coarse, scans are accessed one by one: local thesholding is a matter of resetting thresholds in the fine objects. If these are too large, localisation will be less effective. On the other hand, if the initial threshold is set too high objects will fractionate, but worse than that some parts of the field may be lost entirely. Nevertheless, poor initial thresholding can often be compensated for later at higher computational cost–composite objects which get through the second scan can be split apart, and pieces joined.

A large number of thresholding algorithms have been proposed for this type of problem, mostly based on properties of the density histogram (Prewitt and Mendelsohn, 1966; Weszka, *et al.*, 1974; Weszka, *et al.*, 1973/5). We have found

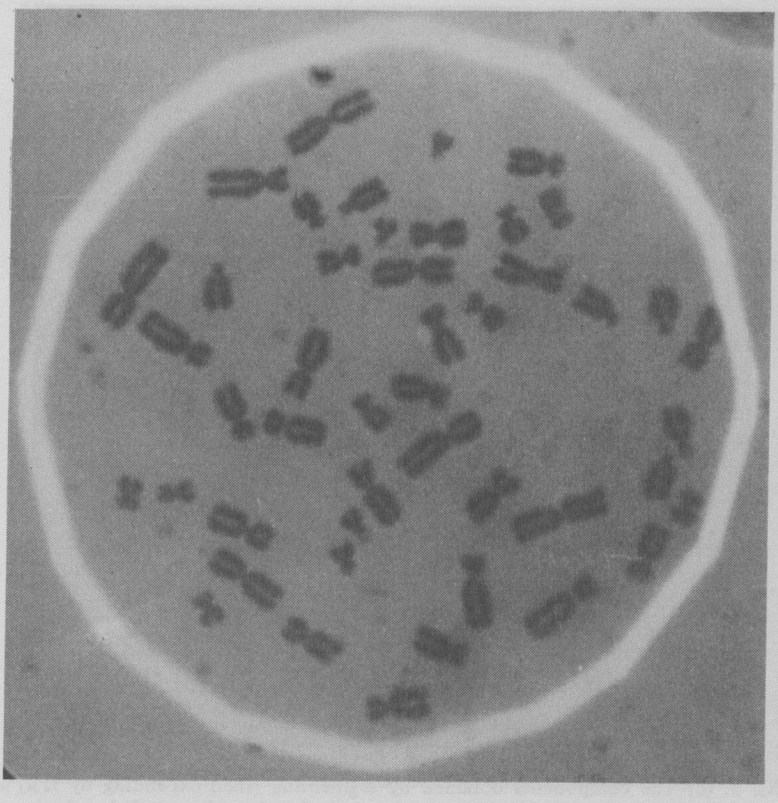


FIG. 3. (b) The scan was carried out at ½ μ resolution.

however that in the wide variety of conditions encountered, particularly the frequent but inconstant presence of extraneous dark or speckled nuclei, a method which simply tries to obtain 40 to 45 objects–the number known to be "right," in the coarse scan field, gives good results.

THRESHOLDING CHROMOSOME FIELDS: A COMPONENT COUNTING ALGORITHM

Thresholding fields of cells or cell-like objects for purposes of counting and measurement is often predicated on the optical density frequency distribution. An example is given in Fig. 3, for a chromosome scan. The large peak corresponds to the background, the small peak to the chromosome bodies, the intervening trough the chromosome edges: this is where the threshold should be set, and Fig. 3c shows the thresholded (and expanded) picture. Fig. 4 shows a histogram with additional and sometimes misleading local maxima and intervening minima due to the presence of parts of an extraneous nucleus in the scanned field.

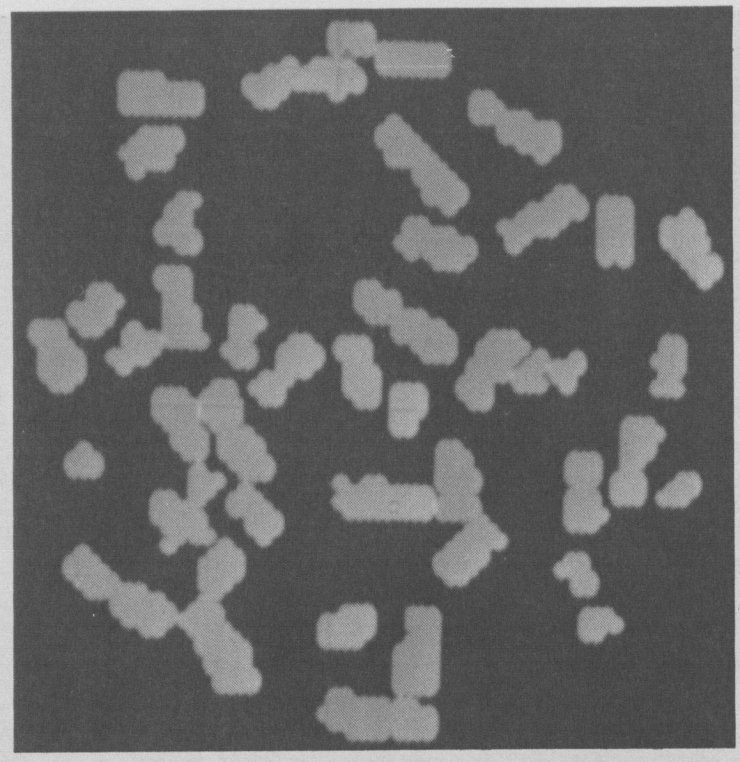


FIG. 3. (c) The components obtained by segmenting (then expanding by ½ μ) at the level shown by the arrow in (a). The occasional overlapping of expanded regions and the small size of others indicates the critical nature of threshold decisions in this low density sampling situation.

Improvements in threshold setting functions in this context usually aim at sharpening the picture maximum and emphasising the intervening edge-related minimum, or use gradients to produce a function which peaks at the optimal separation level.

An example of the first type is

$$f(g_o) = \sum_{g = g_o} \frac{1}{1 + (\underline{\nabla} g)^2} \qquad \text{(I)}$$

i.e., the histogram of density levels inversely weighted by the local point-gradient; this function might be expected to emphasise the level background and foreground "top," and to give low intervening values.

An example of the second type is furnished by

$$f(g_o) = \sum_{g = g_o} |\underline{\nabla} g| \Big/ \sum_{g = g_o} 1 \qquad \text{(II)}$$

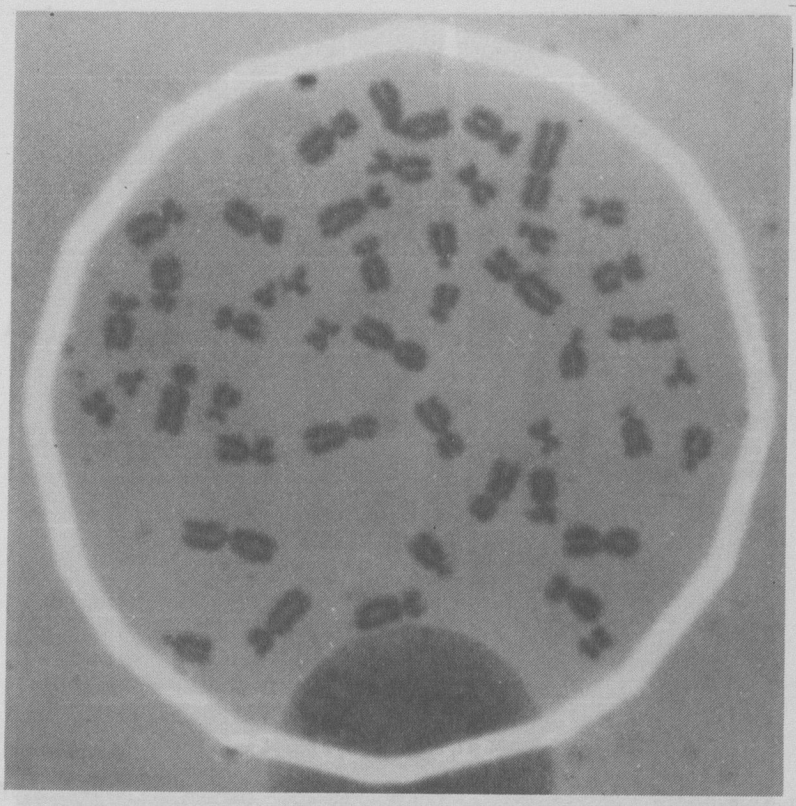


FIG. 4. (a) A field with an intrusive nucleus.

This is the function giving the average gradient value associated with each grey level.

Both of these give useful results on many fields of the sort in which we are interested; but both are liable to gross distortion by the intrusion of extraneous single objects such as the nuclei of cells not in division.

We have therefore sought other approaches, in which the threshold method is more directly related to the objective of segmenting the scene into the 'right' number of constituents.

Crossing numbers

Let g(i,j) be an o.d. field (i = line number, j = column number). We define the (upward) crossing function, $\phi(h,i,j)$ as follows:

$$\phi(h,i,j) = \begin{cases} 1, \text{ if } g(i,j\text{-}1) \leqslant h < g(i,j) \\ 0, \text{ otherwise.} \end{cases}$$

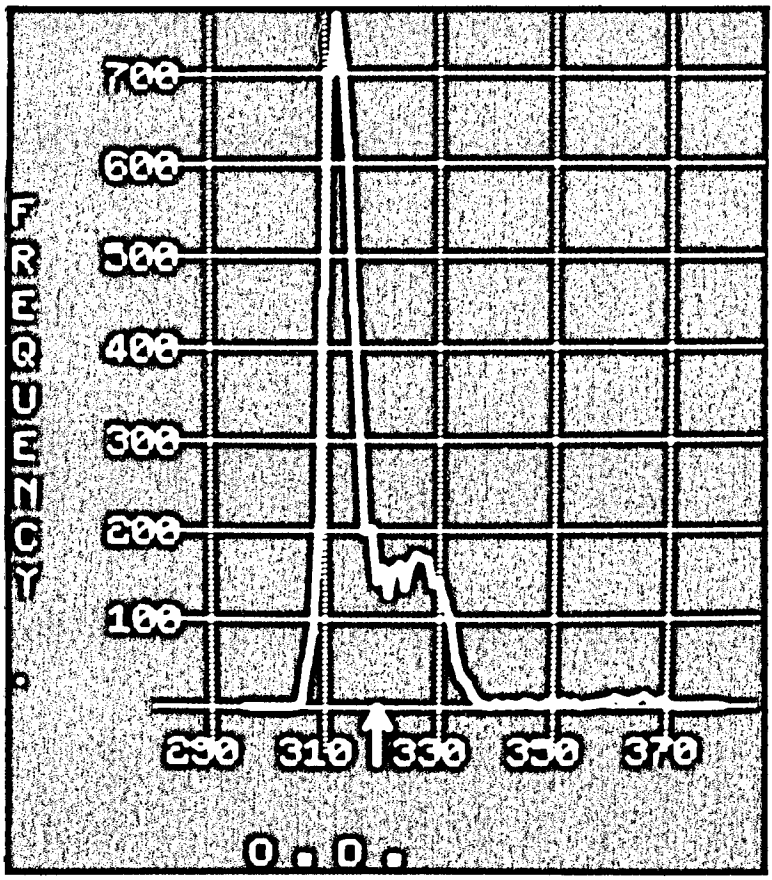


FIG. 4. (b) The histogram is now the superposition of the background, chromosome and nucleus optical density distributions, with a number of distinct local minima and maxima.

The (upward) crossing histogram of g is then

$$f(g) = \sum_{i,j} \phi(g,i,j);$$

i.e., for each level g, f(g) counts the number of left-to-right upward o.d. transitions through g.

To reduce the effects of noise (particularly in the background levels) on gradient and crossing calculations, we can vary these definitions to include a smoothing or trend-seeking element.

For some fixed number of associated elements d let us define

$$\overline{g}(i,j) = \max_{j-d < k \leqslant j} g(i,k),$$

and

$$\underline{g}(i,j) = \min_{j < k \leqslant j+d} g(i,k),$$

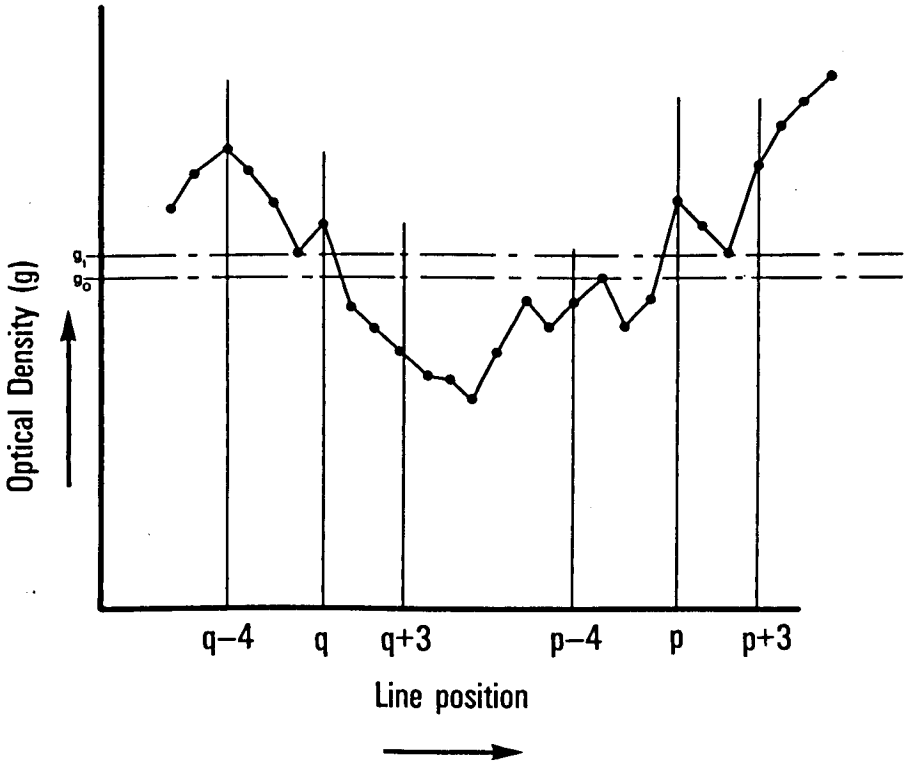


FIG. 5. *Calculation of Crossing Numbers* The diagram illustrates the calculation of $g(i,p) = g_o$ and $g(i,p) = g_1$ in the case d = 4. Crossing numbers of o.d. values between g_o and g_1 will be incremented by 1, providing the similar downward transition at q is within the specified distance, w, of p.

If we now set

$$\phi(h,i,j) = \begin{cases} 1, \text{ if } \overline{g}(i,j) \leqslant h < \underline{g}(i,j) \\ o, \text{ otherwise} \end{cases}$$

and define g as before, transitions will be counted only on genuinely rising edges, not at noisy spots (Fig. 5).

Consider the effect of a single, large, dark artefact or extraneous object in an otherwise more or less homogeneous cellular field: if over a span of d points per line along its edge the level rises monotonically from g_1 to g_2, and there are altogether M points in the edge at each level between g_1 and g_2, the crossing number at these levels will be augmented by M. In the case of the gradient-weighted histogram, the score at each level involved would be augmented by about αM, where $\alpha = \frac{g_2 - g_1}{d}$; since the large α values associated with edges of darkly stained "blobs" do not come into the crossing number calculation we

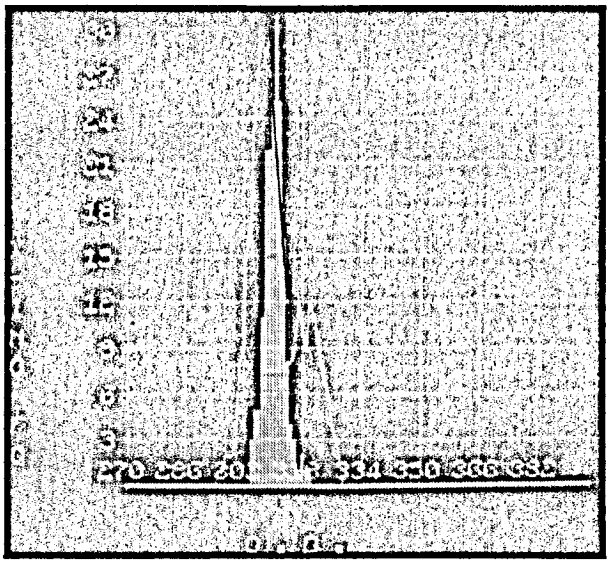


FIG. 6. A crossing frequency histogram. Crossing numbers are shown by the vertical bars, o.d. frequencies by the polygonal line.

should expect less distortion than in the case of the gradient histogram.

The effect of blob-intrusion can be further diminished by use of the consideration that at the levels at which the majority of field constituents separate, downward and upward transitions will alternate within relatively short distances of each other. This is done by modifying the crossing function definitions, thus:

$$\text{Let } \psi(i,j) = \begin{cases} 1 \text{ if } \underline{g}(i,k) > \overline{g}(i,j) \text{ for some } k,\ j\text{-}w < k < j, \\ o \text{ otherwise.} \end{cases}$$

If we now define $\phi'(i,j) = \phi(i,j)\ \psi(i,j)$, ϕ' is 1 if and only if there is a left-max to right-min transition at (i,j) through the o.d. value g, and also, the locally minimised o.d. value has exceeded g at some point in the preceding w points (see Fig. 5). The effect of intrusive dark objects is now minimised (Fig. 6).

However, not all intruders are dark or even uniformly grey—Fig. 4 shows a speckled one, with plenty of structure at intermediate grey levels. Such objects can considerably distort even the modified crossing number histograms. There remains a very direct solution to the problem: if we count the number of objects obtained by cutting at different levels, and then select the threshold which gives the number nearest to that expected the intrusion of one or even several nuclei will not often mislead to a substantial extent. Simple: but at what computational cost?

Note first that to count ends of sequences of touching intervals, at a given cut-level, is cheap enough.

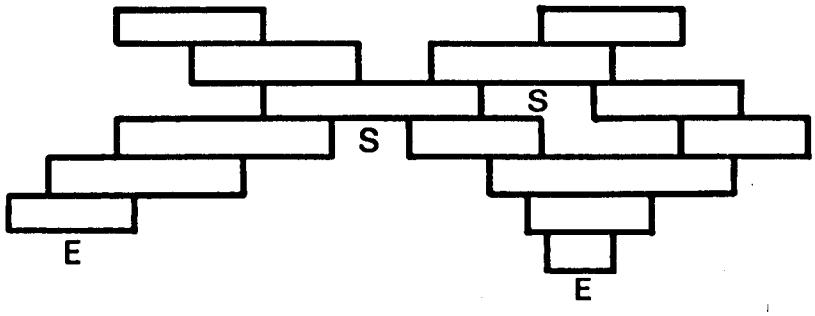


FIG. 7a. *Component Counting* The diagram illustrates a sequence of touching intervals as might be found in successive scan-lines of a digitised image. The single component shown should generate a count of 1 but as there are two splits (S) and two "anti-coincidence points" or ends (E) the count registered would be zero. To correctly count components with holes, as illustrated, a trace of the incidence of intervals upon each other must be kept, in one form or another.

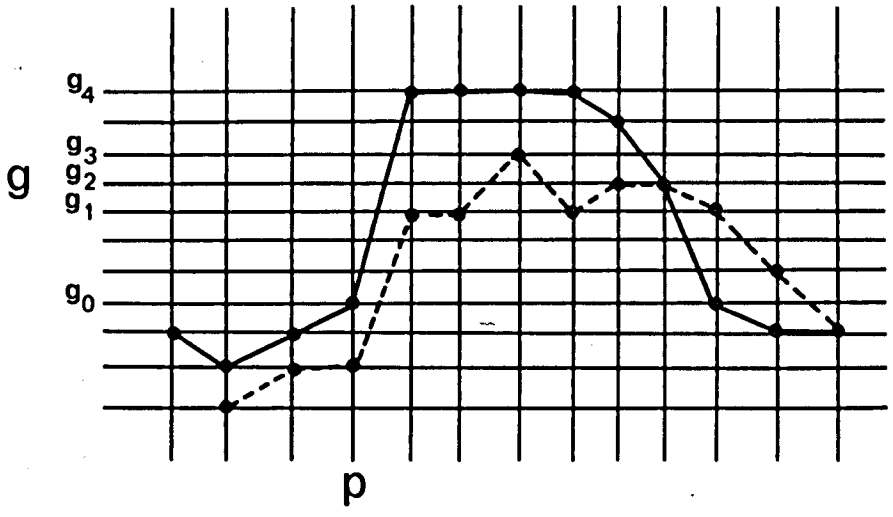


FIG. 7b. The solid and dotted lines denote o.d. value traces in two successive scan lines. At point p the level increases in the first line from g_0 to g_4. For $g_3 < g \leqslant g_4$ the above-g interval beginning at (p+1) has no adjacent successor in the following line, and the component count associated with these levels is increased by 1. For $g_1 < g \leqslant g_2$ there is a split in the following line, so the count is decreased by 1. For other g satisfying $g_0 < g \leqslant g_3$ there is no change in the component count.

Suppose

$$g(i, j\text{-}l) < h \leqslant g(i, j).$$

Let

$$j_1 = \min\ k\text{-}1 \,|\, k > j;\ g(i,k) < h.$$

Then for $j \leqslant j_o \leqslant j_1$ we have $g(i,j) \geqslant h$: i.e., the interval $[j,j_1]$ is above the cutting level h, and the o.d. value drops below h beyond both the left and right hand ends of this interval. Consider the position in the following line. If

$$g(i+1,k) < h \text{ for } j\text{-}1 \leqslant k \leqslant j_1 + 1,$$

we have reached an end of an interval sequence: an anti-coincidence point or A.C.P. and consequently increase the count associated with h, by 1 (Fig. 4b). This will not give an accurate object count: it can be improved by searching for "splits" as well as ends (Fig. 7a), and reducing the count by 1 at each split. To do this, we examine the interval $[j\text{-}1, j_1+1]$ in line $i+1$: if it contains m disjoint segments with o.d. above h we reduce the associated count by m. The object count will now be correct in the absence of holes but there is no computationally cheap way of dealing with holes or other complicated topologies (more to the point perhaps, there is no store-inexpensive way of so doing, remembering that we want an object count at each grey level). Another snag is that we have an object count, but no indication of object size.

How does the count vary with the cutting level? Clearly when the level is very low we have only one object, when very high we have none, and somewhere in between we have the right number. Consider first a field which is uniform except for completely uncorrelated shot-noise. If we threshold at the average field value, there will be a probability of 0.5 that any particular point will be above or below threshold.

Thresholding just below the mean value should give rise to a large number of holes, and therefore to a negative value for the (ACP-split) count. There will be a large number of small noisy components when we threshold just above the mean value, rapidly diminishing as the level increases. Thus in a field having a substantial almost uniform background we should expect first a large negative object count in the background (Fig. 8a) then a large positive count, followed by a small one on the inclined sides of objects and a "correct" count at the right edge (Fig. 8b) level and perhaps a steady value over some density range. Then a rising count as objects break up and finally a decreasing count as they disappear. All these features are shown in Fig. 9a. Since we know in this case how many objects to expect we simply set the threshold at the first value after the post-background trough which gives a count exceeding a small tolerance of the desired number (sometimes, due to poor focusing perhaps, this cannot be attained, and the peak value is used instead). This will not itself yield perfect field separation as

(i) some noise objects will be present
(ii) some of those counted may be intrusive nuclei
(iii) there may be inseparable overlays or touches or separated parts of chromosomes present
(iv) the cell may be abnormal
(v) the count may be falsified by the presence of holes, or effects of shading.

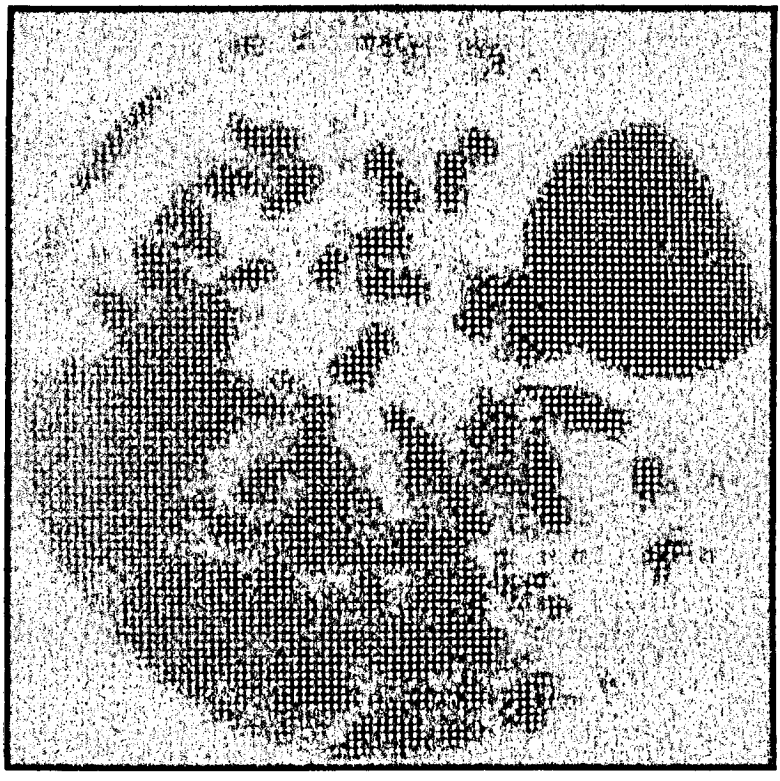


FIG. 8. Components at different threshold levels (cell of Fig. 3b). (a) hole formation is still in progress in the upper left quarter while components separate elsewhere.

However it almost unfailingly segments the field into components of manageable size on which final local segmentation decisions can be made.

The timing (on a 1 μsec. cycle time 18 bit machine) is approx. 100 μsec. per picture element and o.d. value spanned, or about 4 secs. for a 10,000 point picture spanning 40 (relevant) o.d. levels. To keep the level span to a minimum we first generate a straight o.d. histogram of the picture: then develop the object-frequency histogram over the upper quartile of the o.d. range only, as we know that at least ¾ of the field consists of background points.

ASSIGNMENT OF CHROMOSOMES TO GROUPS, USING SIZE AND CENTROMERIC INDEX MEASUREMENTS

A traning set is first selected, comprising perhaps 200 cells from 10 different slides, spanning a range of individuals and cultures. This enables the generation of tables giving for each group of chromosomes recognised and each point of the size/centromeric index plane, the frequency of occurrence of chromosomes with such measurements. These are now smoothed, in effect by replacing each datum by a small 2-dimensional normal distribution, of unit mean, and variance equal

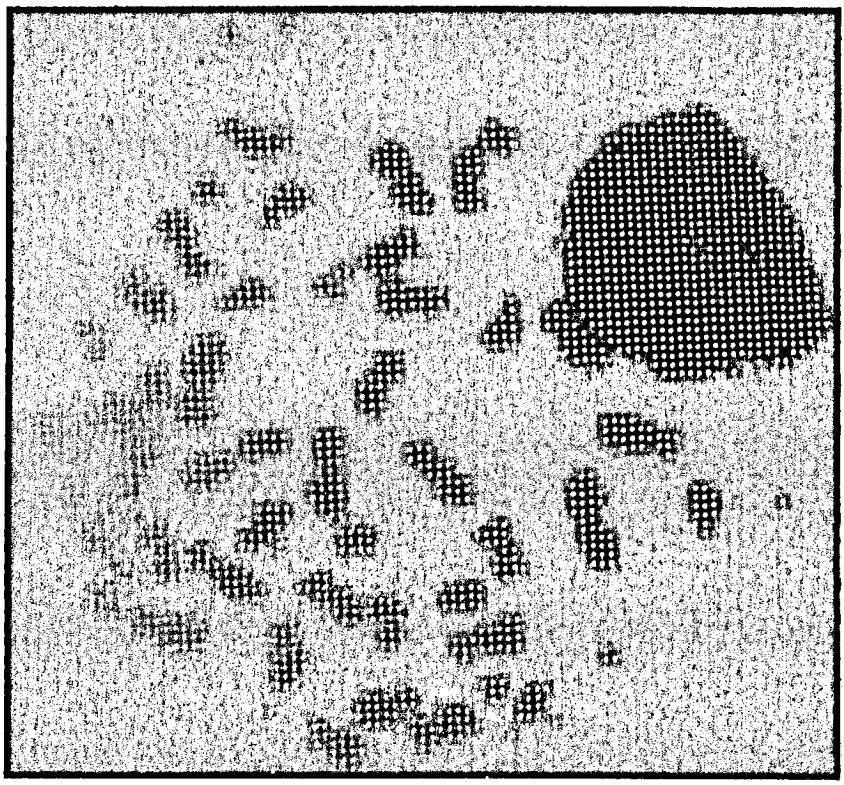


FIG. 8. (b) Segmentation about best possible under the shading conditions present.

to the known r.m.s. measurement error of the system. From the smoothed data a summary table is made, giving for each point in the plane first- second- and third-most likely group assignments and their relative probabilities, and the total relative probability of that point being a measurement from a normal chromosome at all.

This summary table is available to the karyotyping programs during operation. The procedure is as follows. All measurements having been obtained for chromosomes in a cell, each is first allocated to the most likely group. The resulting karyotype is then reviewed. If this is normal (i.e., normal for the sex of the individual if this has been entered) the classification is accepted. If not the group assignments may be revised, providing that these are plausible 2nd- or 3rd-most likely assignments for the chromosomes concerned. An 11th group, the "abnormals" is introduced at this stage. Chromosomes with a very low probability of assignment to any normal group are assigned to this group instead. Since the expected number here is zero, it is always regarded as in excess, unless empty.

However, the problem is more complex than that: to begin with the measure-

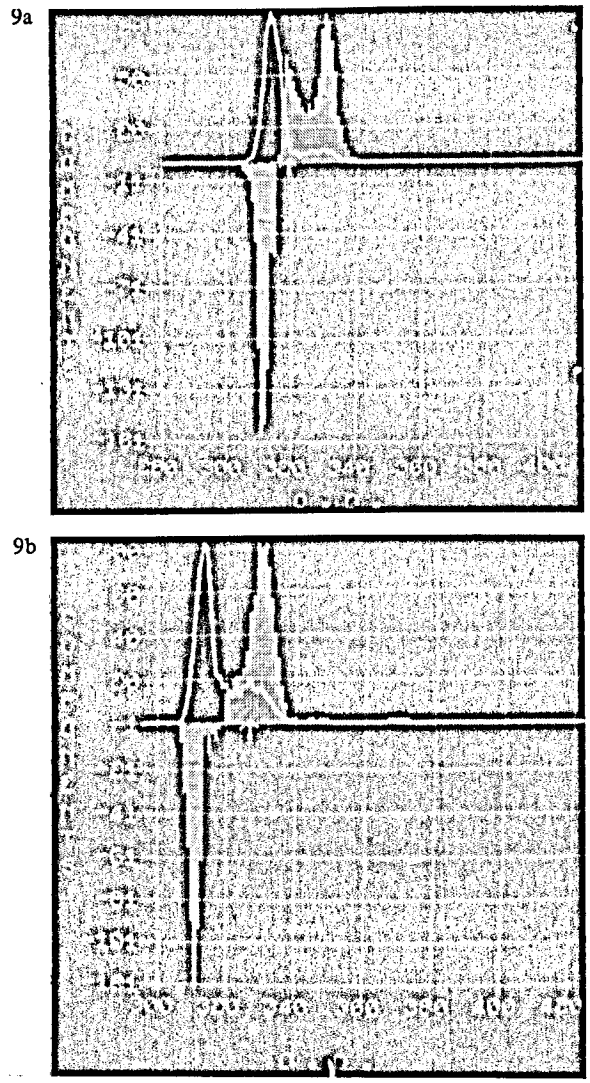


FIG. 9. Component counts (bars) associated with different o.d. levels and, superimposed with different vertical scale, the o.d. frequency histogram. (a) a typical (somewhat shaded) cell (fig. 3b) (b) a cell with intrusive nuclei (fig. 4a).

ments themselves are affected by alterations to the karyotype. To explain why this is so we have to refer back to the question of size normalisation.

Conventional normalisation is done by dividing the size of each chromosome by "adjusted total cell size". This is the sum of the sizes of all chromosomes in the cell, adjusted to allow for any missing or additional ones. This is all very well when the size calculation is done after karyotyping. Before karyotyping is complete however we do not know which ones are missing or additional. In conse-

quence we use an iterative procedure, namely: initially the size is taken to be simply the total size. The first karyotyping assignment is made on this basis. Once a chromosome has been alloocated to a group in which the average size is G, say, in relation to an ideal cell of total size C, assigning a chromosome of area A to this group makes AC/G an estimate of the adjusted actual cell size. By averaging these estimates of total size, for all the chromosomes assigned, (in which the probability of correct assignment is used as a weighting factor) we obtain a more refined estimate. The chromosomes are then classifed again with the new size normalisation, which in turn produces a different normalisation factor. This process is repeated until successive estimates are within prescribed limits of each other. Any subsequent rearrangement of chromosomes will necessitate a reappraisal of the normalising factor as well. Furthermore errors in initial assignment can produce a chain reaction which will result in wrong classification of a number of chromosomes. Suppose, for example, that we are missing a number 1 chromosome (due perhaps to its being involved in a translocation with a small chromosome). If in addition some material is missing from the cell, the normalisation factor will be too small: all chromosome sizes will be over-estimated. Thus perhaps a number 2 will be classified as a 1 on the first pass and there will be a general upward shift of chromosomes not strongly distinguished by centromeric index. The second normalisation will tend to restore the situation, but if it does not, our optimisation procedure may have to make a series of changes, not merely swopping chromosomes from over-full groups directly into deficient ones, but even carrying out cascade transfers into and out of a number of intermediate groups. This leads to the following procedure.

We define the cost of moving a chromosome $\underline{j}$ from group G to group H as

$$\phi(G,H) = -\max\ p(H|j)|j \in H\ ,$$

and consider only those cases for which $\phi(G,H) < \phi_o$, a plausibility constant. Thus $-\phi(G,H)$ is the maximum probability of one of the chromosomes so far assigned to group G belonging, in fact, to group H.

Let $j_{G,H} \in G$ be a chromosome for which $p(H|j_{G,H}) = \phi(G,H)$, i.e., one which realises the minimal cost of a direct transfer from G to H.

The cost $\phi(M)$ of a "cascade-move" $M = [G_o, G_1, \ldots G_{n+1}]$ from $G_o = G$ to $G_{n+1} = H$, moving $j_{G_k, G_{k+1}}$ from G_k to G_{k+1} for $k = 0, 1, \ldots, n$ is defined as $\max\{\phi(G_k, G_{k+1})\}$. That is, the cost of a series of transfers is that of the worst individual case.

For each group G with too many chromosomes we develop all minimal cost (non-cyclic) cascade-move paths M to other groups H. This is done in the following way:

First all (best) possible direct transfers are found.

Next, for each group G with too many chromosomes, we generate an ordered set $\{T_i\} = \{T_i(G)\}$, of terminal points H of paths (cascade transfers) from G, each with an associated path $P = P(G,H)$, as follows: initially let $T_1, T_2 \ldots, T_m$ consist of the terminals of all 1-long paths (direct transfers) from G, with

$P(G,T_i) = [G,T_i]$; if there are none, there is nothing to do. Now set $i = 1$. Examine T_i. If a group Q can be reached by direct transfer from T_i there are two possibilities:

i) $Q \notin \{T_1, T_2, \ldots, T_m\}$. In this case, let T_{m+1} be equal to Q, take P(G,Q) to be $[P(G,T_i),Q]$, (the path obtained by adjoining Q to $P(G,T_i)$) and increment m.

ii) $Q = T_j$, for some j, $1 \leqslant j \leqslant m$. If the existing path P(G,Q) is not more expensive than $[P(G,T_i),Q]$ take no action. Else replace the former by the latter, and *reset* i *to the minimum of* i *and* j (this is the way we make sure that the benefit of reroutings is not lost, i.e., that all possible extensions of 'good' paths are examined).

If $i = m$ we are done; otherwise we step i and examine the next T_i, and so on.

The set $J = \{T_1, T_2, \ldots T_n\}$ thus defined must include all terminal points of possible cascades from G, for if $[G_o, G_1, \ldots G_n]$ is one such, then by definition each G_{k+1} can be reached by direct transfer from G_k. It follows that if $G_k \in J$, also $G_{k+1} \in J$; and certainly $G_1 \in J$.

Further, the associated paths P(G,H), $H \in J$ are minimal. Suppose this is true for all H for which minimal paths of length $\leqslant n$ can be found (for $n = 1$, we are back in the direct transfer case). Then if $[G_o, G_1 \ldots G_n, G_{n+1}]$ is minimal from $G = G_o$ to $H = G_{n+1}$,

$$\begin{aligned}\phi([G_o, G_1, \ldots, G_{n+1}]) &= \max\{\phi([G_o, \ldots, G_n]), \phi([G_n, G_{n+1}])\} \\ &\geqslant \max\{\phi(P(G_o, G_n)), \phi([G_n, G_{n+1}])\} \\ &\geqslant \phi(P(G_o, G_{n+1})).\end{aligned}$$

Thus the proposition is true for terminals with minimal paths of length $\leqslant n + 1$ also, and so for all terminals. If any of the paths from G terminate at a deficit group H, we note it and its cost. When all excess groups have been investigated, the best excess-to-deficit group transfer, if any, is implemented. The procedure is repeated until either there are no excess or no deficit groups left, or until no plausible adjustments can be made.

Since the number of terminal points can never exceed 10, and the number of plausible direct transfers is in practise never more than 5 (usually only 2 or 3), 50 is an approximate upper bound to the number of paths to be considered at any one time out of any one group. This can be increased somewhat by re-routing but the computation time involved remains modest. Iterations of the entire process due to renormalisation can cause this to increase to an undesirably high level however.

CONCLUSION

We have attempted to describe two very different ways in which knowledge of the context in which certain procedures in a problem of image analysis can,

indeed must, be used to modify general strategies. In the one case this is achieved by the use of a simple combinatorial puzzle solving algorithm to improve the results of Bayesian classification. In the other, successive goal-oriented modifications of well-known thresholding algorithms have resulted in a much improved and more reliable initial field segmentation.

ACKNOWLEDGMENT

The ideas discussed here and programs implementing them have been developed over a number of years by the author and colleagues, notably Judith Hilditch, David Mason and Marshall Presser.

REFERENCES

Buckton, K. Personal Communication.

Butler, J.W., Butler, M.K. and Stroud, A. (1963) Automatic analysis of chromosomes, I. *Data Acquisition and Processing in Biology and Medicine*, 3, pp. 261-275. Pergamon Press, New York, NY.

Hilditch, C.J. and Rutovitz, D. (1972) Normalisation of chromosome measurements. *Comput. Biol. Med.*, 2, 167-179.

Mason, D., Lauder, I., Rutovitz, D. and Spowart, G. (1975). Measurement of C-bands in human chromosomes. *Computers in Biology and Medicine*, 5, 1.

Paris Conference (1971). Standardisation in Human Cytogenetics. Birth Defects. Original Article Series, VIII, 7.

Paton, K. (1969) Automatic chromosome identification by the maximum-likelihood method. *Ann. Human Genet.* 33, 177.

Prewitt, J.M.S. and Mendelsohn, M.L. (1966) The analysis of cell images. *Ann. N.Y. Acad. Sci.*, 128, 1035-1053.

Tjio, J.H. and Levan, A. (1965). The chromosome number of man. *Hereditas* 42, 1.

Weszka, J.S., Nagel, R.N., and Rosenfeld, A. (1974) A threshold selection technique. *IEEE Trans. C-23*, 1322-1326.

Weszka, J.S., *et al.*, (1973/5) *TR-243, TR260, TR-336, TR-349*. Computer Science Centre, University of Maryland, 1973/75.

PERCEPTUAL KNOWLEDGE

23

A Duality Concept for the Analysis of Polyhedral Scenes*

David A. Huffman

Board of Studies in Information Sciences
University of California at Santa Cruz

INTRODUCTION

Pictures of scenes containing polyhedra do not record directly the orientations of the various plane bounding surfaces nor the exact locations of the edges and vertices of the objects that are portrayed. An important goal of the analysis of such pictures is to obtain as much information as is possible about these orientations and locations and to record it in a manner that is as easy to understand as is possible. The author introduced a simple means of visualizing this information in an earlier paper (Huffman, 1971). That paper dealt with procedures that could assist one in deciding whether the objects portrayed in certain idealized pictures were realizable or not. The representation that was referred to as a "dual picture-graph" was found to be quite useful as a tool in making these decisions. A more general and powerful representation than the one reported earlier has since been found by the author to be even more useful; for example in a paper ("Realizable configurations of lines in pictures of polyhedra") to be found elsewhere in these proceedings.

The newer representation is an elaboration of the older one and requires a three-dimensional space. Specifically, each member of a given family of parallel planes in the space of the scene corresponds to a unique point in the space of the dual-scene, each line in the scene corresponds uniquely to a line in the dual-scene, and each point uniquely to a plane. The duality concept proposed here is completely symmetric. That is, the mapping of planes, lines, and points of the scene onto the points, lines, and planes of the dual-scene is exactly the same as the one that maps these same entities in the dual-scene onto their correspondents in the scene.

If orthographic projections are made of the objects in the (x,y,z) coordinate system of the *scene* and of the corresponding objects in the (u,v,w) coordinate system of the dual-scene, the *picture* and dual-picture, in the (x,y) and (u,v)

*The research reported in this paper was supported by the National Science Foundation under Grant GJ-28451.

coordinate systems, respectively, result. These two pictures have precisely complementary roles: the orientations of planes and slopes of lines, that are not directly recorded in one representation, are recorded in the dual representation, and vice versa.

Several examples of the use of the dual-scene are included in this paper. Finally, an extension of the duality concept that is appropriate for more general surfaces in reported briefly.

DEFINITION OF THE DUAL OF A PLANE

We distinguish between the three dimensional space of the *scene* ((x,y,z) coordinates) and the three dimensional space of the *dual-scene* ((u,v,w) coordinates). Similarly we distinguish between the two dimensional space of the *picture* ((x,y) coordinates) and the two dimensional space of the *dual-picture* ((u,v) coordinates) that are obtained by orthographic projection. Thus, in the picture information about z values is unrecorded and in the dual-picture information about w values is unrecorded. Furthermore, we adopt the convention that z (and w) increase in the direction away from the viewer. Because we have assumed that pictures and dual-pictures are obtained by orthographic projection there is no loss of generality in taking the plane z=0 to be the picture plane and the plane w=0 to be the dual-picture plane.

The entity dual to the plane z=ax+by+c in the scene is defined to be the point at u=a, v=b, w=-c in the dual-scene. As a consequence all planes parallel to a given plane have common values of the parameters a and b, but differing values of the parameter c. The corresponding set of distinct points in the dual-scene all project onto the same point (u=a, v=b) in the dual-picture. It also follows that the infinite set of planes having the point x=0, y=0, z=c in common maps onto the infinite set of points contained in the plane w=-c.

In the dual-picture the distance from the origin to the point u=a, v=b equals the tangent of the angle between the corresponding scene plane and the plane z=0. Furthermore, the direction from the origin to the point u=a, v=b is indicative of the direction in which the scene plane tilts. For example, when a is positive and b is negative the value of z on the plane in the scene will increase (corresponding to an increase in range from the viewer) when x is increased and y is decreased.

Alternately, the point at u=a, v=b (and w=0) in the dual-picture can be thought of as the point of intersection between the plane w=0 and a line passing through the point u=0, v=0, w=1 when that latter line has the direction that is normal to the family of scene planes represented by the given point. (Note also the discussion associated with Figure 9, following.)

THE DUALS OF SCENE LINES AND SCENE POINTS

Consider the intersection of the two planes defined by

$$z = a_1x + b_1y + c_1$$
$$\text{and } z = a_2x + b_2y + c_2$$

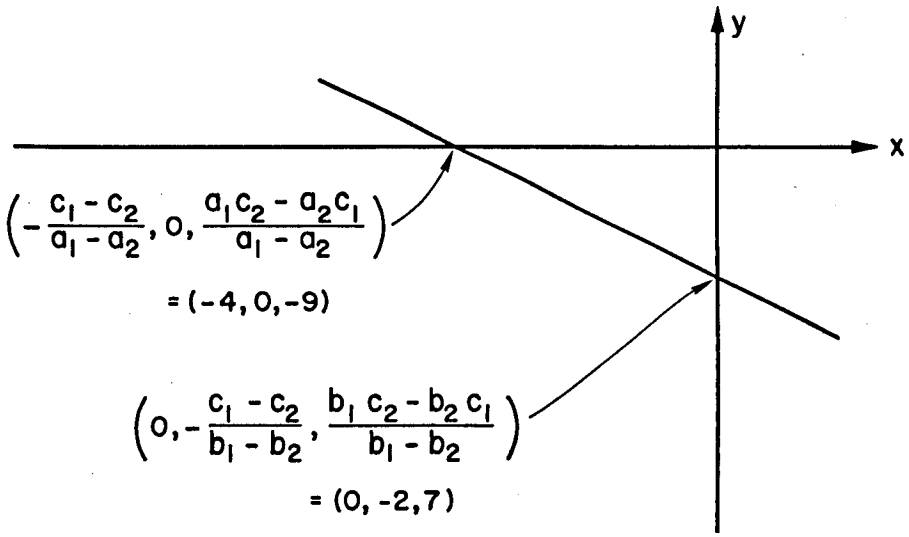


FIG. 1. A picture line.

The resulting line satisfies the equations

$$0 = (a_1 - a_2)x + (b_1 - b_2)y + (c_1 - c_2)$$
$$(a_1 - a_2)z = (a_1 b_2 - b_1 a_2)y + (a_1 c_2 - c_1 a_2)$$
$$(b_1 - b_2)z = (b_1 a_2 - a_1 b_2)x + (b_1 c_2 - c_1 b_2).$$

An example of this intersection line, drawn to scale for the case $a_1=3$, $b_1=-2$, $c_1=3$, $a_2=2$, $b_2=-4$, $c_2=-1$ is shown in the picture of Figure 1. We note for future reference that the length of the *picture line* contained between the x-axis and y-axis intercepts is

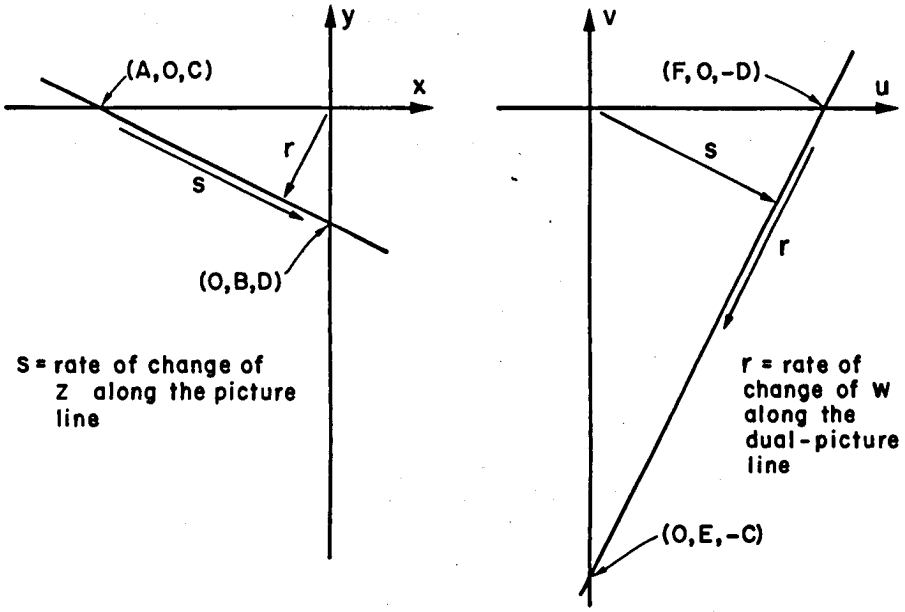
$$\left| \frac{(c_1 - c_2)\sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}}{(a_1 - a_2)(b_1 - b_2)} \right|$$

This length is $2\sqrt{5}$ for our example.

Also, the change in z between the corresponding two points on the *scene line* has the magnitude

$$\left| \frac{(c_1 - c_2)(a_1 b_2 - b_1 a_2)}{(a_1 - a_2)(b_1 - b_2)} \right|$$

This change in z is 16 for our example.



(a) the picture

(b) the dual-picture

$$AF = -BE = C - D$$

$$r = \frac{AB}{\sqrt{A^2 + B^2}} = \frac{D - C}{\sqrt{E^2 + F^2}}$$

$$s = \frac{-EF}{\sqrt{E^2 + F^2}} = \frac{D - C}{\sqrt{A^2 + B^2}}$$

(c) relationships among line parameters

FIG. 3. Illustrating relationships between a scene line and a dual–scene line.

set of three planes in the scene has the defined dual relationship to the plane determined by the three corresponding points in the dual-scene. Therefore the duality is complete: plane to point, line to line, and point to plane.

To a given line in the scene there corresponds a uniquely determined line in the dual-scene. The line in the dual-scene can be thought of as comprised of an infinite set of points each element of which represents one of the set of planes containing the given line in the scene. Alternately, the line in the dual-scene is contained in all those planes of the dual-scene that correspond to the infinite set of points that constitute the given line in the scene. Similarly, to a given point in the scene there corresponds a uniquely determined plane of the dual-scene. Each point of that plane corresponds in the scene to one of the infinite set of planes

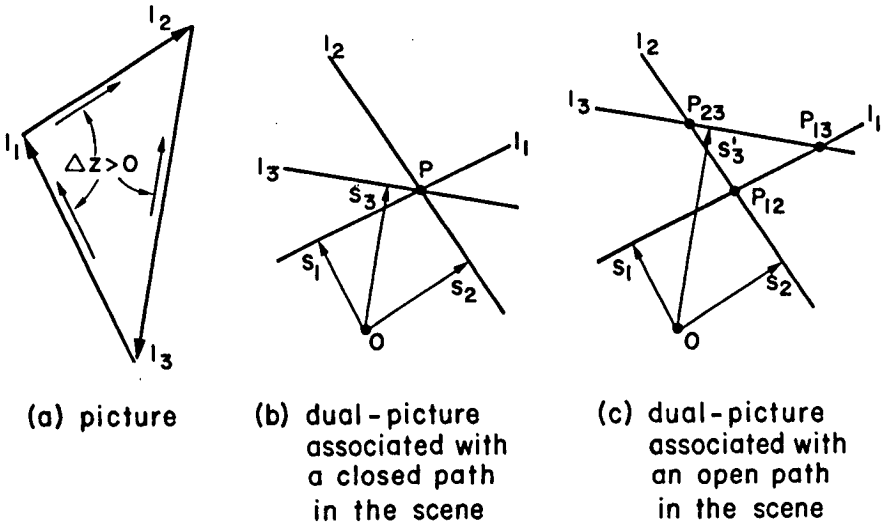


FIG. 4. The analysis of an apparently closed path in a picture.

that contain the given point. Because of the completely symmetric relationship between the scene and the dual-scene each of the statements above is valid when the words "scene" and "dual-scene" are interchanged.

CONDITIONS FOR CLOSURE OF A SEQUENCE OF LINE SEGMENTS IN THE SCENE

Consider the closed sequence of three oriented line segments (hereafter called simply "lines" when no ambiguity will result) shown in Figure 4-a. It is assumed that all of these lines lie in a common plane, P. In the dual-picture of Figure 4-b the corresponding three lines will all pass through the point P associated with the given plane. The placement of that point with respect to the origin of the dual picture tells us that in this example the plane P in the picture is tilted up and to the right. The distance from the origin to the point in the dual-picture equals the tangent of the angle of that tilt.

The "slopes" of the three lines in the picture are equal to the distances to the three lines in the dual-picture. Thus the total amount by which z changes in traversing the line ℓ_i is equal to the signed product $\ell_i s_i$ where ℓ_i and s_i are the length and slope of the line ℓ_i. (Where no confusion will result we shall use "ℓ_i" to refer both to the i^{th} line segment and to its length.) We observe that z increases as we move clockwise along the first and second lines and decreases as we transverse the third line. Because the net change in z around the closed sequence of picture lines must be zero it follows that $\sum_i \ell_i s_i = 0$. We also note that this result is independent of the position in the dual-picture of P relative to 0.

If we modify the dual-picture so that all three lines do not intersect at a

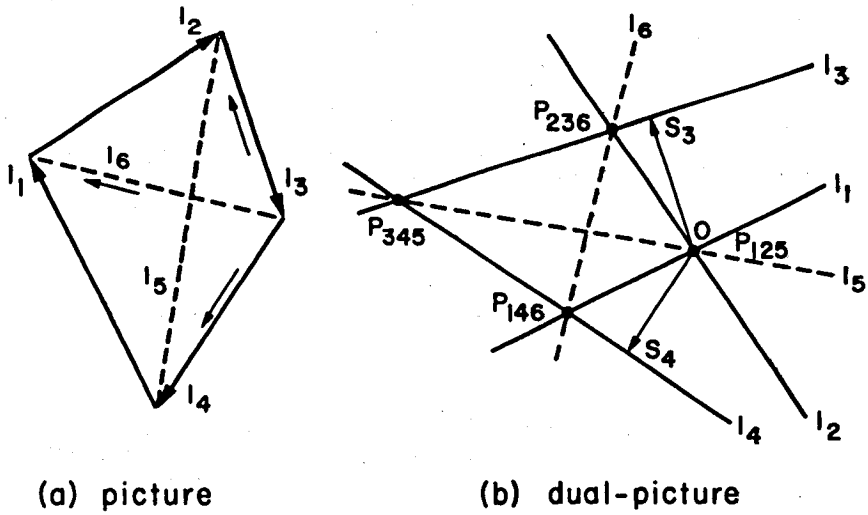


FIG. 5. The representation of a closed picture path.

common point (for instance, by moving ℓ_3 as shown in Figure 4-c) three different points are determined instead of only one and it becomes clear that the expression $\sum_i \ell_i s_i$ is no longer equal to zero. This means that the three lines that apparently form a closed path (in the picture) do not actually form a closed path in the scene itself. In our example the modification we have made corresponds to increasing the magnitude of the slope of the third picture line. If we were to assume, for instance, that the upper end of the third picture line remained fixed then at the lower end of that line we know that the first and third lines only appear to intersect. The point P_{13} in the dual-picture then represents all those planes that are parallel to both ℓ_1 and ℓ_3.

A more complicated example of an apparently closed path of four lines is shown in the picture of Figure 5-a. There are many possible ways of drawing the associated four lines in the dual-picture so that each is at right angles to the corresponding picture line. Not all of these give a set of slopes corresponding to a closed picture path. One that does is shown in Figure 5-b. If for convenience we choose the intersection of ℓ_1 and ℓ_2 to be the dual-picture origin we note that the net change in z around the picture path will be zero only if $s_3/s_4 = -\ell_4/\ell_3$. Thus the relative lengths of the line segments in the dual-picture are fixed if we want that representation to depict a closed picture path. It is easy to prove that when the appropriate proportions are present the line ℓ_5 in the picture will be at right angles to its correspondent in the dual-picture. A similar comment holds true for the line ℓ_6.

CONDITIONS AMONG THE LINES OF A CUT-SET

The examples of the previous section illustrate constraints in a dual-picture

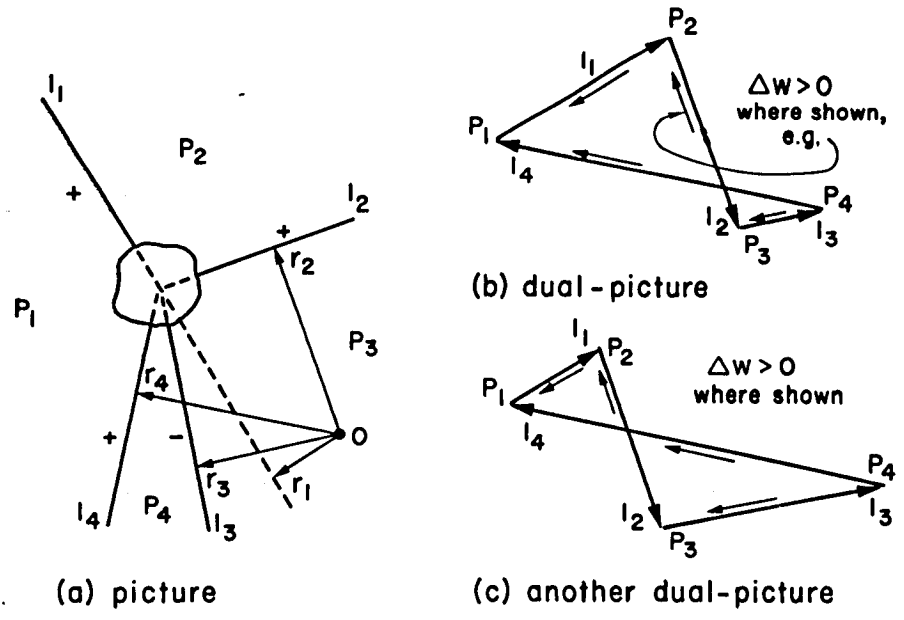


FIG. 6. A single-vertex cut-set.

that correspond to conditions for the closing of a *path* of lines in the scene. In this section we illustrate constraints that correspond to the topologically dual concept: a *cut-set* of lines. Here we consider a cut-set of picture lines to be all those lines that enter a simple closed region of the picture from outside that region. The region can be that around a single vertex and the lines of the cut-set those that depict all the (visible) edges incident at that vertex, as in Figure 6-a. More generally the region can include more than one vertex and the lines of the cut-set may have arbitrary positions, as is illustrated in Figure 7-a.

In Figure 6-a we assume that the four edges portrayed are incident at the single vertex. The line-labels indicate that two of the edges are "convex" (+) and one is "concave" (-). A clockwise encircling of the region associated with the cut-set corresponds in the dual-picture to a closed path along the directed lines of the dual representation. There are many different sets of four planes that could be present in the scene that would yield the given picture. For any one of these interpretations the corresponding points in the dual-scene are well-defined (and lie in a common plane). It necessarily follows that the net Δw around the closed path in the dual-scene is zero and, therefore, that $\sum_i \ell_i r_i = 0$. One possible choice of picture origin is shown in the figure. Two possible choices of orientations of the planes that are depicted in the picture would give the dual representations of Figure 6-b and 6-c.

We note that the issues raised in the example of Figure 6 are analogous to those raised in Figure 4. A closed path of lines and associated set of intersection points all contained in a single plane in one representation correspond in the

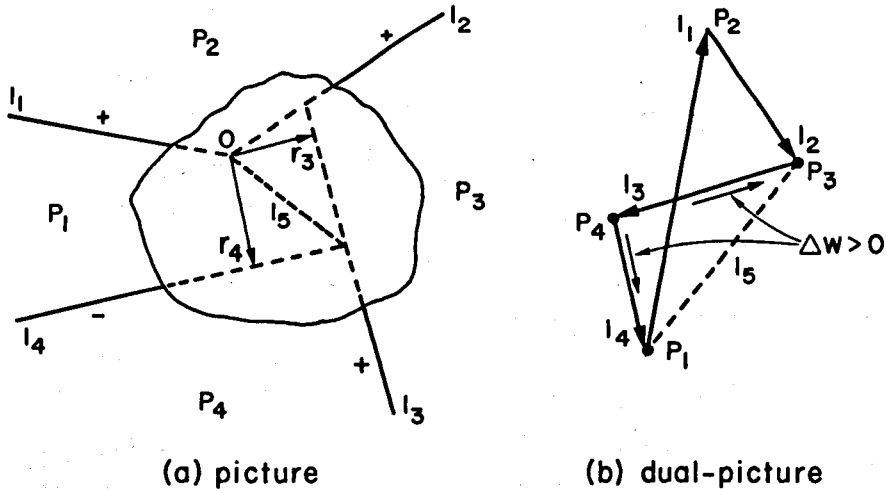


FIG. 7. A multi-vertex cut-set.

other representation to a set of lines and associated planes that all contain a single point.

The example shown in Figure 7 is more general than the previous one in that the four planes depicted in the picture cannot all contain a common vertex. The four lines of the dual-picture must, of course, be orthogonal to their correspondents in the picture. In this new example the ratios between pairs of lengths of dual-picture lines cannot be chosen arbitrarily. This can be easily seen if, for instance, we choose the picture origin to be the point shown. The reason for making this particular choice is that the distances from the origin to the third and fourth lines will then be zero. It is clear that then the ratio of the magnitudes of r_1 and r_2 is fixed. In order that $\sum_i \ell_i r_i = 0$ the ratio of the magnitudes of ℓ_1 and ℓ_2 in the dual-picture must be the one shown in the figure.

The dual-picture of Figure 7-b has the correct proportions. As a verification of this contention we note that one possible completion of the picture is that shown by dotted lines. The fifth line (common to planes P_1 and P_3 in the picture) is perpendicular, as is required, to its correspondent in the dual-picture. The points a and b in the scene correspond to the two planes determined by P_1, P_2, and P_3 and by P_1, P_3, and P_4, respectively, in the dual-scene.

The issues raised in the examples of Figures 5 and 7 are analogous. In each case in one representation a cyclic sequence of planes and the set of lines formed by the intersection of planes that are pairwise adjacent in that sequence correspond in the other representation to a cyclic sequence of points and the set of lines determined by points that are pairwise adjacent in that sequence.

AMBIGUITY OF SCALE AND ORIGIN

A picture inherently contains two kinds of ambiguity. Information about the

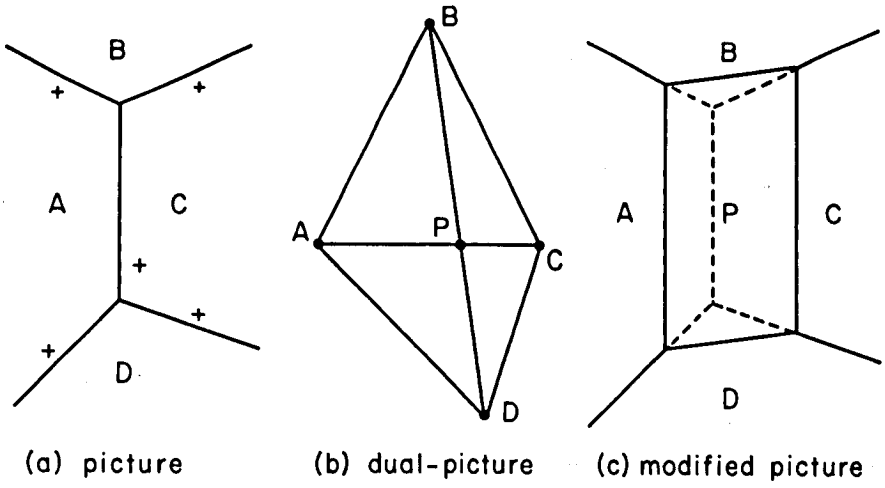


FIG. 8. An origin- and scale-independent problem.

depth of the object(s) depicted is lost because of the necessary projection. The visible planes of a polyhedral object may all have orientations that are nearly alike; that is, the object may be "shallow". On the other hand the same picture may be of a "deep" object, in which case the directions in which the normals to the various planes point may widely differ. These two situations may be represented by two different dual-pictures. The shallow object would have a dual-picture in which the various points (associated with the directions taken by the normals to the scene planes) were all close together. The points in the dual-picture for a deeper object would be further apart.

By changing the scale of a dual-picture we can represent either of the situations above. For instance, multiplying the lengths of all lines in a dual-picture by a factor $k > 1$ corresponds to multiplying by k the slopes of all lines in the scene. Similarly, multiplying the lengths of all scene lines by k corresponds to multiplying the slope of all dual-scene lines by k. It is easy to see that a "size" change in an object in the scene would cause a proportionate change in the recorded picture but no effect in the dual-picture. A change in depth of the scene object would cause a change in the size of the dual-picture portrayal even though no change would be apparent in the picture.

Even when the sizes of the picture and dual-picture images are determined another kind of ambiguity still remains. The distance from the dual-picture origin to each point in the representation is associated with the (tangent of the) angle between the corresponding scene plane and the reference plane z=0. Each possible position of the dual-picture origin corresponds to a different set of orientations of the scene planes. And yet each such possibility can correspond to the given picture. Similarly, a change in position of the picture origin is associated with a change in the orientations of the planes of the dual-scene.

A valuable feature of the picture/dual-picture pair of representations in

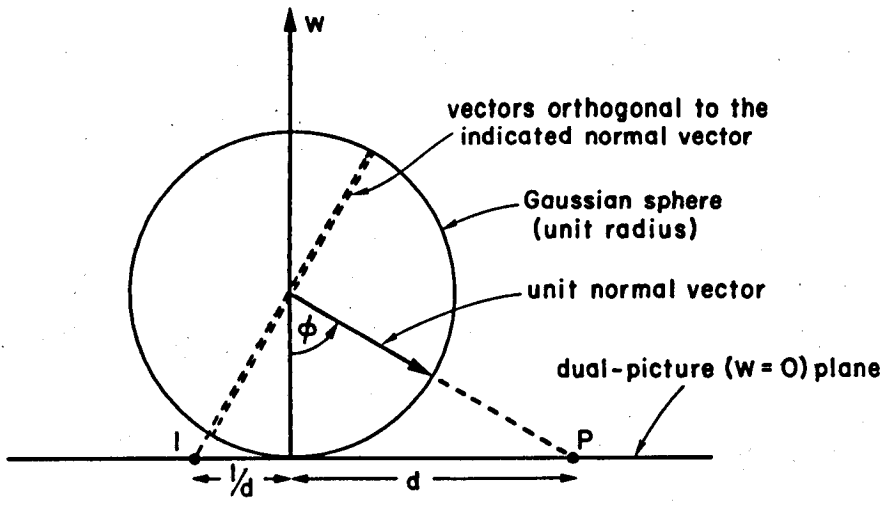


FIG. 9. Representation of planes orthogonal to a given plane.

dealing with many analysis problems is that often we need not concern ourselves with the scale or placement of the origin in either the picture or dual-picture. Consider for instance the (partially complete) picture of Figure 8-a. All five lines have labels indicating that the edges of the object are convex toward the viewer. How can we slice through the object with a plane P so that the new surface that results has the shape of a parallelogram?

We find that orientations of the lines of the dual-picture are determined even though its origin and scale are not specified (see Figure 8-b). If the line common to A and P and the line common to C and P are to be parallel (in the scene and in the picture) the corresponding dual-picture lines must be parallel or must be the same line. Similarly, the lines common to B and P and to D and P will be parallel or the same line when the corresponding dual-picture lines are parallel. Clearly the point P shown in the dual-picture is the only possible one. By noting the orientation of the line BPD we determine (by perpendicularity) the orientation of the lines common to B and P and to D and P (see Figure 8-c).

The size of the parallelogram is not determined by the problem statement. Any of an infinite number of planes parallel to P would also give parallelogram slices through the object. All such planes are represented by the same dual-picture point.

REPRESENTATION OF ORTHOGONAL PLANES

There are many situations in which it is important to be able to represent the set of those planes that are orthogonal to a given plane. In this section we will discuss this representation and give examples of its use.

Consider a given plane P in the scene. Let the direction of the normal vector be that shown in Figure 9. Let the base of that normal be at the point u=v=0,

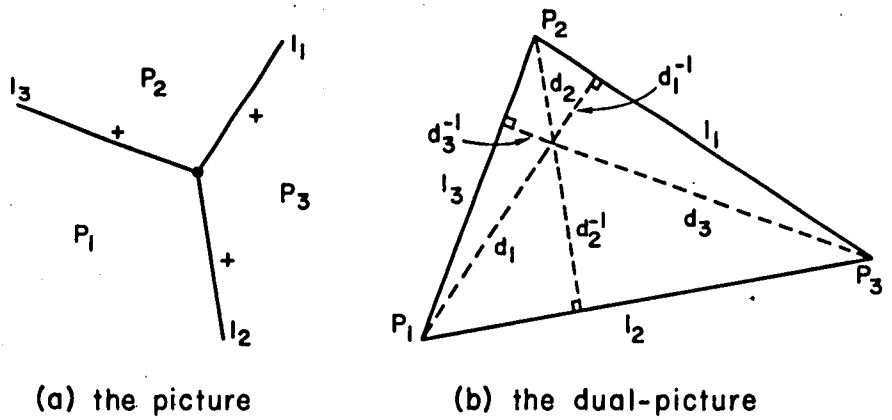


FIG. 10. Representation of a cube vertex.

w=1 and its intersection with the w=0 plane be at u=a, v=b at distance $d = \sqrt{a^2 + b^2}$ from the origin. The angle ϕ between the plane P and the picture plane (z=0) is the angle between the normal vector described above and the w-axis. As we have commented earlier $d = \tan \phi$.

It is convenient in dealing with normal vectors to consider the unit radius "Gaussian sphere" and the end points of those vectors on that sphere (Hilbert and Cohn-Vossen, 1952). In this case the normals to planes orthogonal to P are represented on a great circle of the sphere 90° away from the point associated with P. The corresponding points in the dual-picture plane are found on the line ℓ, the end-view of which is shown in the figure. The distance from the origin to this line is 1/d.

An example of the use of representations of orthogonal planes is given in Figure 10. The picture given is assumed to be that of a cube vertex. The dual-picture has the constraints that are indicated in Figure 10-b. In particular note that in that representation a line ℓ_i represents all those planes that are orthogonal to plane P_i. The three dotted lines through the origin are perpendicular to ℓ_1, ℓ_2, and ℓ_3 in the dual-picture and therefore parallel to their correspondents in the picture. It is apparent that if the picture of the edges of a cube is given and if the orientation of one plane (say P_1) in the scene is given the orientations of the other two planes is uniquely determined. This fact is apparent in the dual-picture as well.

Another example of the representation of orthogonal planes is given in Figure 11. The problem posed is as follows. In the picture two lines ℓ_1 and ℓ_2 are given, each with a specified slope. The separation (change in the value of z) at their apparent intersection is also given. The picture of their common perpendicular is to be determined.

The specification of the slopes s_1 and s_2 determines ℓ_1 and ℓ_2 in the dual-picture. Their intersection point P corresponds to the set of planes that are parallel to both ℓ_1 and ℓ_2 in the picture. The desired perpendicular in the scene is

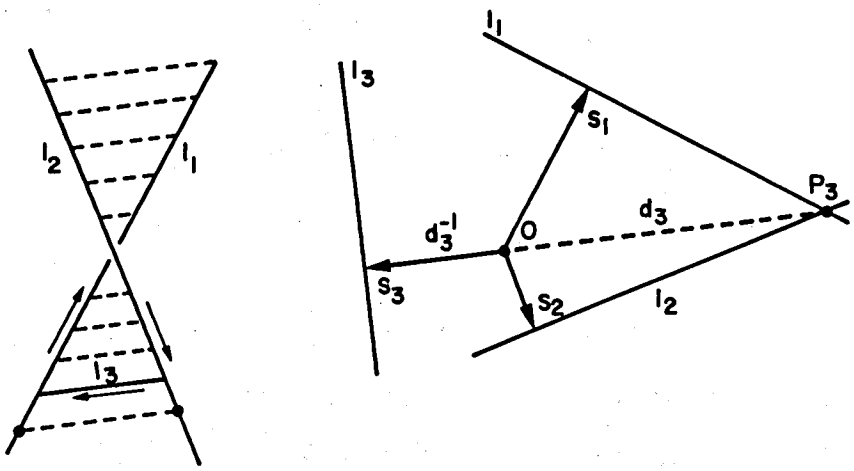


FIG. 11. Determination of a common perpendicular.

a line common to a set of planes all of which are orthogonal to the set corresponding to P. Thus the line ℓ_3 in the dual picture represents this perpendicular. Its slope is determined by the specifications of the problem. All of the dotted lines in the picture have a direction consistent with that of l_3 in the dual-picture. It is apparent that since the slopes of ℓ_1, ℓ_2, and ℓ_3 are now all determined there is a unique choice of ℓ_3 in the picture that will satisfy the given change of z at the apparent intersection of ℓ_1 and ℓ_2.

THE DETERMINATION OF SHADOWS

Another application of the representation of orthogonal planes is in the determination of shadows. Consider the dual-picture of Figure 12. The point S represents the "sun" (assumed to be infinitely distant from the objects in the scene). The distance d is the tangent of the angle between the vector pointing to the sun and the normal to the reference plane. For instance, if d were equal to zero the sun would be directly overhead for an observer standing on the reference plane. The line ℓ_s represents all those planes that have normals that are at right angles to the vector pointing to the sun. Planes such as those corresponding to the points A and B that are on one side of ℓ_s have normals with angles less than 90° from the sun vector. Planes such as C and D on the other side of ℓ_s have associated angles that exceed 90°. Therefore A and B are oriented so that the sun illuminates them and C and D are shadowed. For an observer standing on a plane such as E that is represented by a point on ℓ_s the sun would appear to be on the horizon.

Consider now the simple configuration of three planes and associated edges depicted in Figure 13-a. The labels on the lines indicate a (convex) "crest" line shared by planes B and C and two other (concave) lines where those two planes

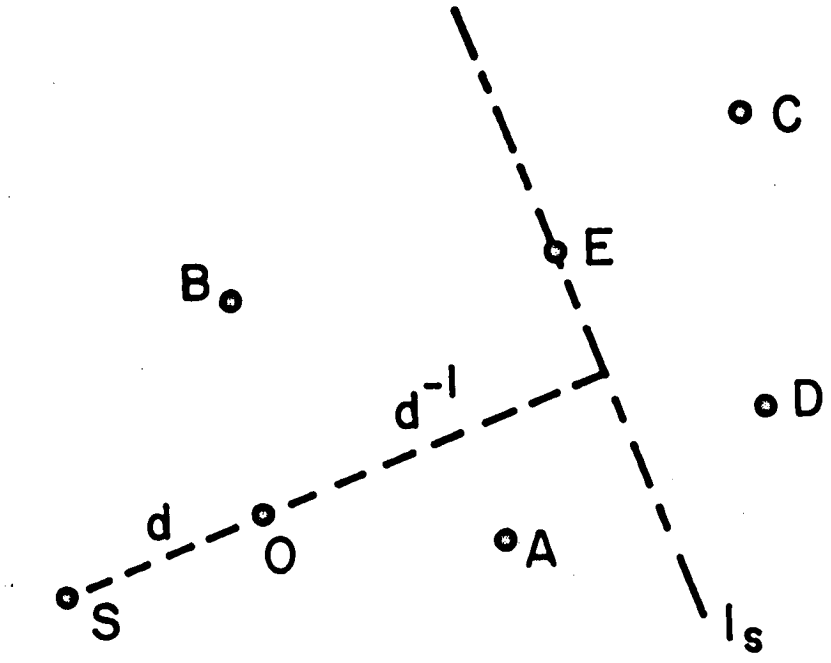


FIG. 12. Representation of the "sun" in the dual–picture.

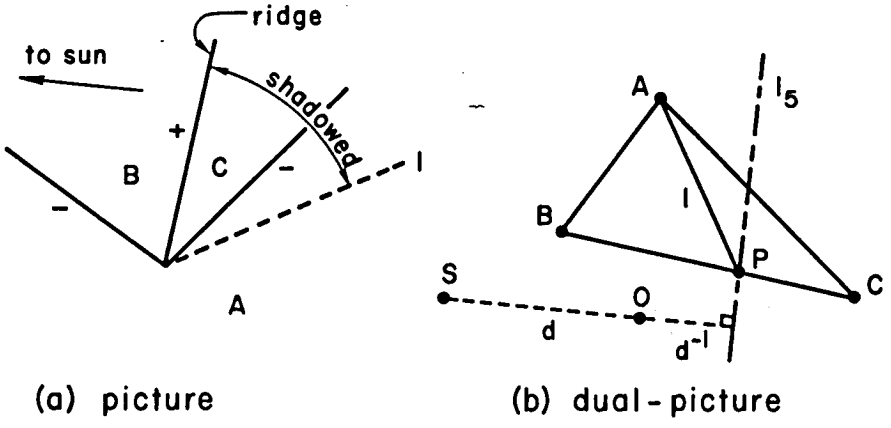
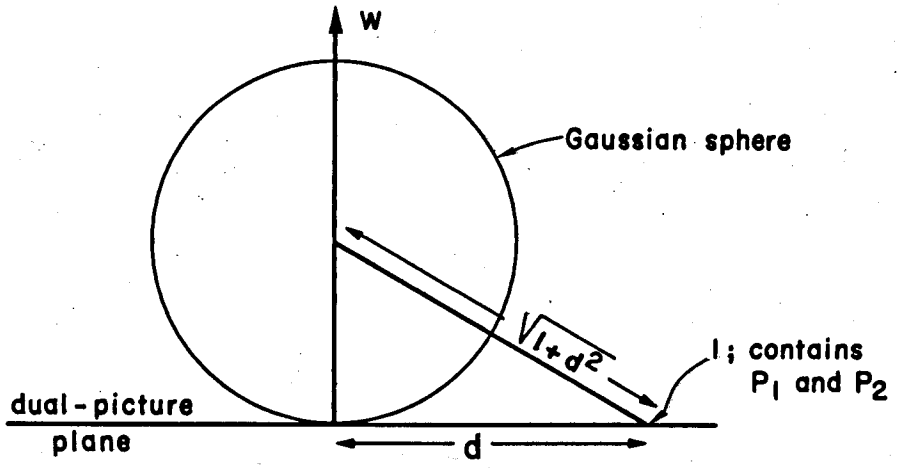


FIG. 13. Determination of the shadow of a line on a plane.

intersect the plane A. We assume that the orientations of these planes correspond to the points A, B, and C shown in the dual-picture. We also assume that the sun vector is the one shown. The line ℓ_s is constructed as indicated. Since S and C are on opposite sides of that line the plane C is turned away from the sun. It is apparent that under these conditions the crest-line will cast a shadow on plane A.

In order to determine the shadow-line ℓ cast by the ridge we first observe that



(a) edge view of the dual-picture plane

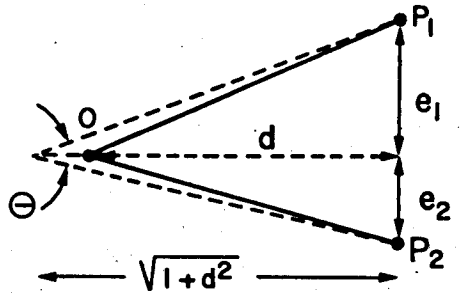


FIG. 14. Construction for finding the angle between two planes.

the ridge and ℓ determine a plane P that must be represented somewhere on the line ℓ_s in the dual. Because P contains the line common to planes B and C its dual-picture representation must also be somewhere on the line BC. The intersection of ℓ_s and the line BC is the necessary location of the point P. Because the shadow-line ℓ is in plane A as well as in plane P it must in the picture be perpendicular to the corresponding dual-picture line that contains the points A and P.

DETERMINATION OF THE ANGLE BETWEEN TWO PLANES

It is convenient to be able to find the angle between two planes from their representations in the dual-picture. The construction shown in Figure 14 illustrates one method of achieving this result. In Figure 14-b the representations for two planes P_1 and P_2 are given. A perpendicular from the origin to the line ℓ containing P_1 and P_2 has a length d and divides that line into two segments of lengths e_1 and e_2. If we were to look at the dual-picture edgewise and from the direction parallel to ℓ the resulting view would be that given in Figure 14-a.

The true angle between the two planes is the angle between their normals. This angle is that between P_1 and P_2 (in the dual-picture plane) as seen from the center of the Gaussian sphere. The distance to ℓ from the sphere center is $\sqrt{1 + d^2}$. Therefore the construction shown in dotted lines in the dual-picture gives the true angle, θ, between the two planes.

The angle θ can also be found from the expression

$$\theta = \tan^{-1} \frac{e_1}{\sqrt{1 + d^2}} + \tan^{-1} \frac{e_2}{\sqrt{1 + d^2}}$$

If P_1 and P_2 lie on the same side of the perpendicular to ℓ from 0 the values of e_1 and e_2 will have opposite signs and the expression must be evaluated accordingly.

BRIEF SUMMARY AND A GENERALIZATION OF THE DUAL-SURFACE CONCEPT

The preceding sections of this paper have dealt with a concept of duality that was introduced for the purpose of analyzing pictures of scenes containing plane-bounded objects (polyhedra). This new concept is one that treats the "third" dimension of the scene or of the dual-scene in a manner that is distinct from the manner in which the other two dimensions are treated. The resulting picture and dual-picture pairs contain complementary types of information in the sense that depth information missing in one representation is present in the other representation, and vice versa. A generalization of this duality concept is reported briefly below. The author will elaborate on this generalization in a future paper.

We consider two surfaces defined by

$$z = f(x,y) \text{ and } w = g(u,v)$$

to be dual if for each point and associated tangent plane on one there is a corresponding tangent plane and associated point on the other. For the point (x_o,y_o,z_o) on the first surface we define the corresponding tangent plane to be $w = x_o u + y_o v - z_o$ on the second. Similarly for the point (u_o,v_o,w_o) on this second surface we define the corresponding tangent plane to be $z = u_o x + v_o y - w_o$ on the first surface. Thus for the pair of points of interest we have $w_o + z_o = x_o u_o + y_o v_o$. The point on one surface will be associated with the direction of the normal to the tangent plane on the other when

$$u_o = \frac{\partial f}{\partial x}\Big]_{x_o, y_o} \qquad v_o = \frac{\partial f}{\partial y}\Big]_{x_o, y_o}$$

$$x_o = \frac{\partial g}{\partial u}\Big]_{u_o, v_o} \qquad y_o = \frac{\partial g}{\partial v}\Big]_{u_o, v_o}$$

As a simple example consider the surface $z = f(x,y) = x^3 + xy$. We have

$$u = \frac{\partial f}{\partial x} = 3x^2 + y$$

$$v = \frac{\partial f}{\partial y} = x$$

These equations have the solution

$$x = v,\ y = u - 3v^2$$

Therefore
$$\begin{aligned} w &= ux + vy - z \\ &= (3x^2 + y)\,x + xy - (x^3 + xy) \\ &= 2x^3 + xy \\ &= 2v^3 + v(u - 3v^2) \\ &= -v^3 + uv \end{aligned}$$

Therefore the surface that has the desired dual relationship is given by $w = g(u,v) = uv - v^3$.

Finally, we observe that it is possible for a surface to be *self-dual*; that is, for the functions f and g to be the same. As an example, the reader can verify that

$$z = f(x,y) = \frac{(x^2 - y^2)\cos\theta \pm 2xy \sin\theta}{2}$$

describes a self-dual surface; in this case a saddle surface. The corresponding surface in the other coordinate system is

$$w = g(u,v) = \frac{(u^2 - v^2)\cos\theta \pm 2\,uv \sin\theta}{2}$$

The self-duality is independent of the value of the parameter θ.

REFERENCES

Huffman, D.A. (1971) "Impossible Objects as Nonsense Sentences." *Machine Intelligence* 6, (eds. B. Meltzer and D. Michie), Edinburgh University Press, Edinburgh.

Hilbert, D. and Cohn-Vossen, H. (1952) *Geometry and the Imagination*, Chelsea Publishing Co., New York.

24

Realizable Configurations of Lines in Pictures of Polyhedra†

David A. Huffman

Board of Studies in Information Sciences
University of California at Santa Cruz

INTRODUCTION

In an idealized picture of a scene that contains only polyhedra each line segment that is recorded can have only one of four possible "meanings". In order to understand the picture it is necessary that we be able to label each line with one of the four corresponding labels: +, -, → or ←. A "+" or "-" label is associated, respectively, with a convex or concave edge that has both of its two associated planes visible. A line labelled with an arrow refers to a convex edge oriented so that only one of these two planes is visible from the camera and the other is hidden behind it. The orientation of the arrow along the line is such that the planes are to the right of the arrow. If no consistent set of line labels is possible the picture is of an "impossible"-object. If one or more labellings are possible a necessary condition for the picture to be realizable will have been satisfied and the picture may, indeed, be ambiguous. An earlier paper (Huffman, 1971) dealt with the restricted case of scenes containing only trihedral objects. This paper generalizes the results of the earlier one to the case of scenes that contain polyhedra having arbitrary numbers of planes associated with the vertices.

The catalog of the twelve possible pictures of trihedral vertices is shown in Figure 1. When polyhedra with arbitrary numbers of edges incident at each vertex are considered the development of an extended catalog is clearly impossible; it would need to contain an infinite number of entries. What is needed instead is a decision procedure that can be applied to any configuration of labelled lines that is a candidate for inclusion in that hypothetical catalog.

We shall also generalize this procedure still further so that it applies not only to the configurations of lines incident at a single picture node, but also to the situation in which an arbitrary number of lines having arbitrary labels enter a

†The research reported in this paper was supported by the National Science Foundation under Grant GJ-28451.

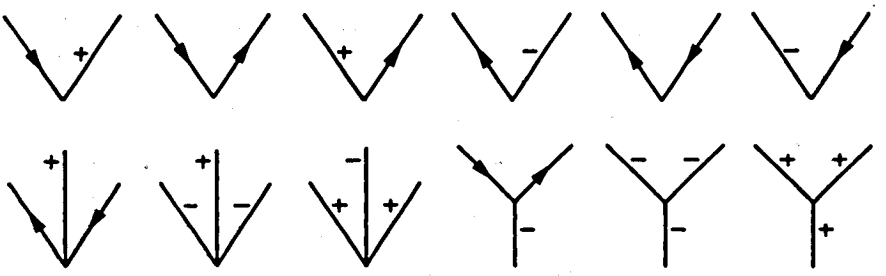


FIG. 1. Catalog of labelled-line configurations possible at nodes in pictures of trihedral objects.

closed picture region from arbitrary directions. The procedure also indicates, if the configuration is realizable, how the planes bounded by these lines can be oriented.

A by-product of the effort required in developing the main theme is a list of all possible interpretations of picture configurations of the form "T". For the trihedral case such a configuration could not correspond to a physical vertex and could only be interpreted as evidence that one edge was obscured by another.

In the research reported here use is made of the concept of the *dual-scene* and the *dual-picture* reported in another paper in this volume. Briefly, the entity dual to plane $z = ax + by + c$ in the scene is defined to be the point at $(u,v,w) = (a,b,-c)$ in the dual-scene. The corresponding projected point in the dual-picture is at $(u,v) = (a,b)$. The intersection of two planes in the scene determines a line that has a correspondent in the dual-scene. This latter line can be thought of as comprised of an infinite set of points each of which represents one of the set of planes containing the given line in the scene.

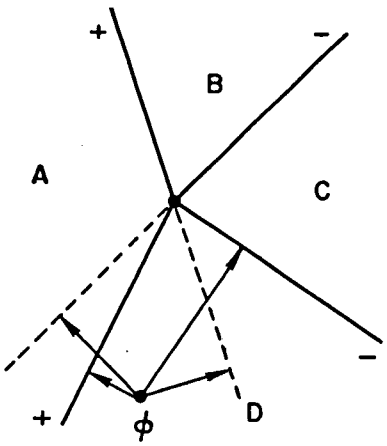
A useful property of such a dual pair of lines is that the rate of change of z along a given scene line with respect to motion along the projected line in the (x,y) picture plane is equal to the distance from the origin in the (u,v) dual-picture plane to the corresponding dual-scene line that is projected upon it. Similarly, the rate of change of w along a given dual-scene line with respect to motion along the projected dual-line in the (u,v) dual-picture plane is equal to the distance from the origin in the (x,y) picture plane to the scene line that is projected upon it.

A most delightful consequence of this definition of dual is that a dual pair of lines (in the scene and dual-scene) are recorded in the picture and dual-picture by lines that are at right angles to each other.

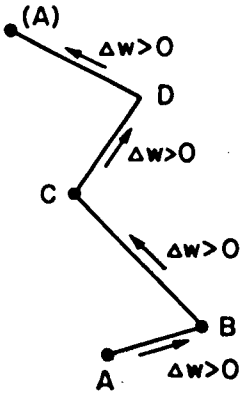
Finally, a point in the scene corresponds in the dual-scene to the plane of all points dual to planes containing the given point of the scene. Specifically, the point at $(x,y,z) = (d,e,f)$ corresponds to the plane $w = du + ev - f$.

CONFIGURATIONS OF LINES INCIDENT AT A SINGLE PICTURE NODE

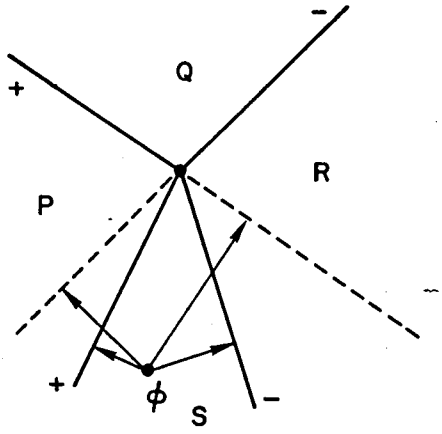
As we move along any path on the surface of a polyhedron there is generated



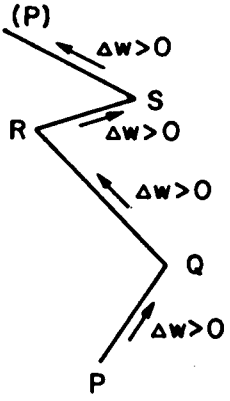
(a) picture of a non-realizable configuration



(b) the corresponding trace in the dual-picture



(c) picture of a non-realizable configuration



(d) the corresponding trace in the dual-picture

FIG. 2. A pair of equivalent non-realizable configurations and their dual–pictures.

a corresponding path in the dual-scene (and in the associated dual-picture). We shall call this latter path the *trace* of the original path. The trace will consist of an ordered sequence of points (associated with faces of the polyhedron) joined by lines (associated with edges of the polyhedron). We shall be interested in this paper in traces associated with closed contours drawn on the polyhedral faces in the scene. Such a closed contour may enclose just one polyhedral vertex, or it may enclose more than one. In either case since a contour is closed the corresponding trace must also be closed (if the polyhedron is to be realizable). Note, for example, that a trace may appear to close in the dual-picture but not

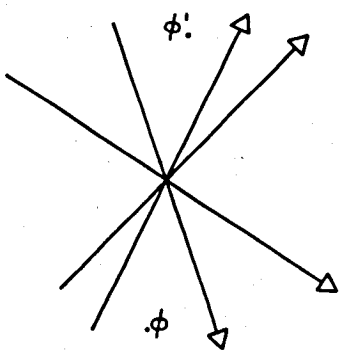


FIG. 3. An equivalent representation of the configurations of Figure 2.

actually close in the dual-scene. For instance, if a trace starts at a point (u,v,w) = (a,b,c) but ends at (u,v,w) = (a,b,c') then the scene path begins and ends not on the same plane but on two different (parallel) planes.

A configuration of labelled lines crossed by a contour in a picture will be termed *realizable* if and only if it can be demonstrated that the corresponding trace can be closed (in the dual-scene). In other words, a configuration of picture lines is realizable when and only when there exists at least one orientation of each of the associated planes such that their intersections would yield the lines of the given configuration.

We shall first deal exclusively with the issue of the realizability of configurations of lines that are all labelled "+" or "-". It will then be easy to modify our results so that they apply to configurations that contain arrow-labelled lines.

Consider first the nodal configuration shown in Figure 2-a. The lengths of the line segments of the trace of Figure 2-b are not determined by the picture, but their directions are. It is apparent that the trace cannot be closed because each portion of the trace has a positive v-component. Hence the configuration is not realizable.

Alternatively, one can observe that if we were to choose the origin of the picture at the point ϕ the directions from there to each of the four lines are those shown. These same directions are those in which w increases along the associated segments of the trace (Huffman, 1976). Thus for that choice of origin, not only does the projected trace not close in the (u,v) plane, but neither can the net change of w along the trace be zero, as required.

The configuration of Figure 2-c is related to the example above in that certain of its lines are given the other sign-label and "*reversed*" (that is, extended on the opposite side of the given node). The resulting trace has components that take the earlier directions although they appear in a different order. The second configuration is thus also not realizable.

It is apparent that *any* of the configurations that result from reversing lines is unrealizable. The arrangement of *oriented lines* of Figure 2-e then can be used for all such configurations. The direction indicated by an open arrowhead is the

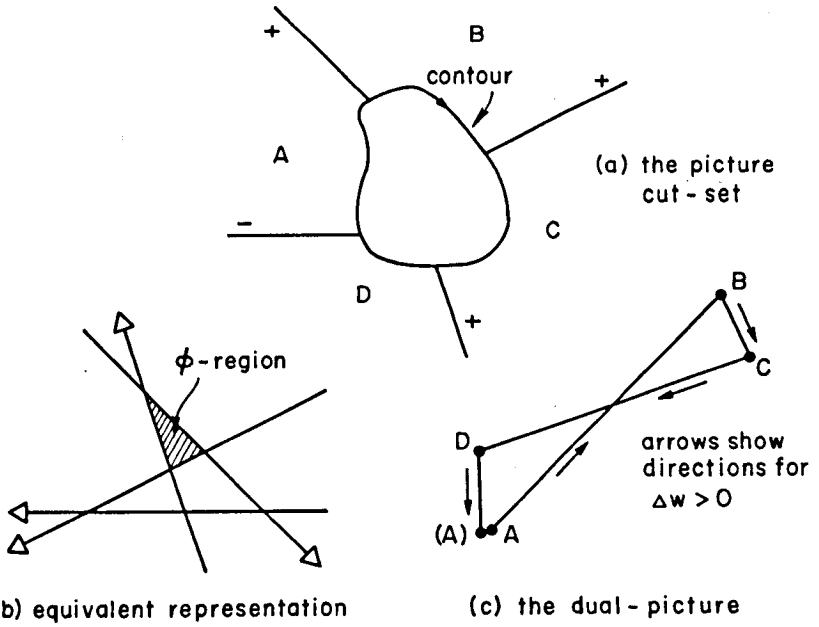


FIG. 4. An unrealizable cut-set.

same as that that a "+" labelled line would take as it entered the region enclosed by the contour and opposite to that that a "-" labelled line would take. The direction in which w increases along the trace is obtained from that directed line by rotating it 90° counterclockwise. The fact that there exists a possible location ϕ that is to the right of all directed lines in the picture then guarantees that the net change in w along the trace would be positive, rather than zero. Similarly, the origin ϕ' is to the left of all directed lines and would lead to a net negative change in w. (Note that the directed line segments designated by *open* arrowheads are not the same as the line *labels* that have been used earlier.)

Still another method of understanding the non-realizable configuration of Figure 2-a is to imagine piercing the scene at the location ϕ. If each of the four planes depicted were extended then along the ray ϕ from the viewer toward the scene the distance from the observer to the A-plane would be greater than to the B-plane (because their common edge is labelled "+" (convex) as shown). That is, $z_A > z_B$. Similarly $z_B > z_C$. Because the line common to planes C and D is concave we can conclude that at ϕ the distance to C is greater than that to D; that is, $z_C > z_D$. Similarly $z_D > z_A$. It is however *impossible* that $z_A > z_B > z_C > z_D > z_A$. Such a relationship is an example of what we shall call here a *cyclic inequality*. The existence of a point ϕ (or ϕ'), at which a cyclic inequality exists, can be used to prove that the configuration is not realizable.

We see from the preceding examples that only the orientations of the directed lines in the equivalent configuration of Figure 3 are important in determining the realizability of the configuration(s) that it replaces.

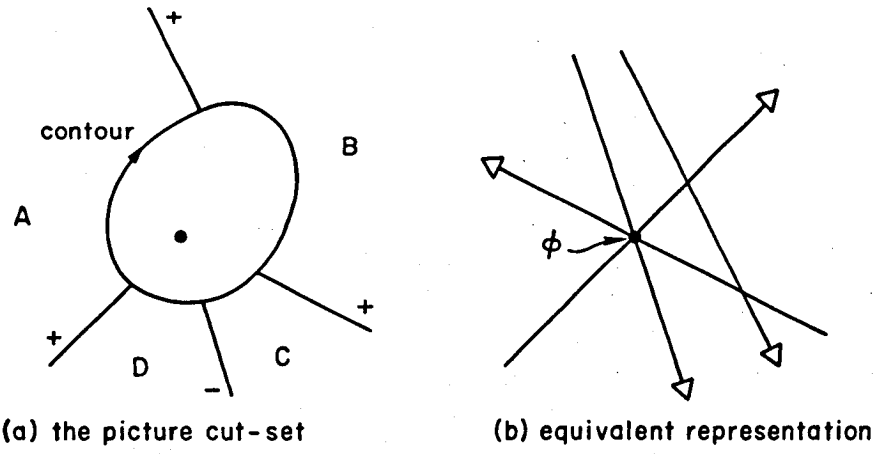


FIG. 5. A cut-set having a single ϕ - point.

CONFIGURATIONS OF LINES NOT ALL INCIDENT AT A SINGLE PICTURE NODE

As an example of a contour around a region into which picture lines enter from more general directions consider Figure 4-a. (Obviously two or more nodes would be necessary inside the contour.) The equivalent oriented line representation is given in Figure 4-b. A region to the right of all lines is indicated. That region is thus the location for possible ϕ points any one of which could be used to demonstrate that the configuration is not realizable.

The dual-picture in Figure 4-c shows a trace that would seem at first glance to be closed. If we recall that the directions in which w increases can be found by turning the corresponding oriented lines 90° counterclockwise we discover that the trace only appears to close. Alternately we can demonstrate a cyclic inequality at any one of the ϕ-points. Because we can now predict that the increments of w along a trace will be uniformly positive (or uniformly negative) when the existence of a ϕ- (or ϕ'-) point can be established we shall no longer bother in these cases to attempt to demonstrate candidates for closed traces.

We shall now give an example of a non-realizable configuration for which there is an isolated ϕ-point. In Figure 5 a cut-set is demonstrated having the property that all but one of the four lines is incident at a single point. The equivalent picture of oriented lines indicates that a ϕ-point (the only possible one) exists at the location common to three of the lines. At that ϕ-point it can easily be shown that $z_A > z_B = z_C = z_D = z_A$. Obviously this is an impossibility.

It is now possible to give a general *definition of a ϕ-point (ϕ'-point)* based on representations using oriented lines:

> A ϕ-point (ϕ'-point) of an arbitrary cut-set of lines in a picture of polyhedra is a point that is to the right of (left of) some line of the cut-set and that is not to the left of (right of) any other lines.

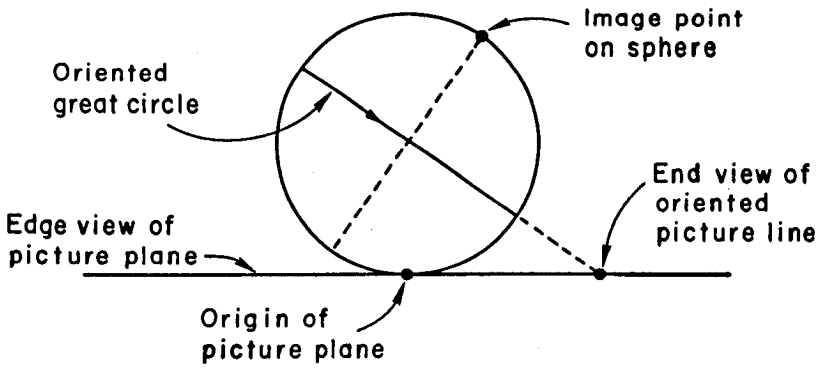


FIG. 6. Correspondence of image point to an oriented picture line.

We have demonstrated that a ϕ- (or ϕ'-) point implies that a cyclic inequality exists and thus that the cut-set of lines is not realizable. We shall prove below that a cut-set for which there is no ϕ- (or ϕ'-) point is realizable.

The major steps in our proof are the following: we will first show, for realizable cut-sets of +/- labelled lines entering a region of a picture in such a way that no three or more lines have a common intersection, that if there is no ϕ- or ϕ'-point there must exist some subset of four lines of the cut-set for which there is also no ϕ- or ϕ'-point. Secondly, we will show that this basic cut-set of four lines does have a realization. Finally, we will indicate that the other lines of the original cut-set can be added back to the cut-set without affecting the issue of realizability. It will be apparent in the development of the proof how to deal with the special case of a cut-set that has all of its lines incident at a single point.

A "SPHERICAL" REPRESENTATION OF PICTURE LINES

During the first part of the proof we shall use an alternate representation of the oriented picture lines. Imagine (see Figure 6) a unit radius sphere tangent to the picture plane at its origin. An oriented picture line and the center of the sphere determine a plane that intersects the surface of the sphere in a great circle that has an orientation that corresponds to that of the oriented line. In Figure 6 we have assumed that the oriented line is pointed into the paper. This oriented great circle, in turn, determines (using a "right-hand rule") the direction of a unit *normal vector* and a corresponding uniquely determined *image point* on the sphere. It is a straightforward matter to show that lines that pass through a common picture point correspond to images that lie on a common great circle.

A fact that is especially important for our purposes is that if a set of picture lines all pass some point in the picture plane clockwise (or all pass counter-clockwise) their images all lie in some common hemisphere. We use this fact to establish that when there exists a ϕ- (or ϕ'-) point for a cut-set of picture lines we can determine some corresponding great circle and hemisphere such that all normal vectors terminate either in the hemisphere or on the great circle boundary of that hemisphere. It is easy to show also that an appropriate

boundary of that hemisphere is that great circle that is orthogonal to the line that contains the sphere center and the ϕ- (or ϕ'-) point of the picture plane.

We can see therefore that when and only when there is *no* ϕ- or ϕ'- point can positive linear combinations of the normal vectors that correspond to the picture lines span the three-dimensional space of the sphere. It is apparent that a minimum of four lines is necessary to assure that there is no ϕ- or ϕ'- point. Furthermore, if a cut-set of more than four lines has no such point a subset of exactly four of them can be found that also has that property.

The result above could be stated more formally in terms of the language of convexity theory, with heavy reliance on Carathéodory's Theorem. This theorem states that if S is a set of points in n-dimensional Euclidean space and if P is a point of the convex hull of S then P is in the convex hull of some set consisting of n+1 or fewer points of S. The convex hull of a set is the smallest convex set that contains the given set. Alternately it is the set of points of the form $\Sigma\alpha_i P_i$ for points P_i of the set and non-negative weights α_i with property $\Sigma\alpha_i = 1$.

For the special case in which all the picture lines pass through a common point the appropriate statement is that no ϕ- or ϕ'- point exists if and only if positive linear combinations of the associated normal vectors span the *two*-dimensional space associated with their common great circle. In this special case three (rather than four) vectors are necessary and sufficient. If more than three vectors are in the set and if their positive linear combinations span the associated two-dimensional space then some subset of exactly *three* vectors can be found that have that same property.

THE BASIC REALIZABLE CUT-SET OF FOUR LINES

For the second part of our proof (that non-existence of a ϕ- or ϕ'- point implies realizability of the cut-set) we need to show that a four line cut-set having no such point is realizable. It is easy to show that four oriented picture lines having this property must necessarily form the patttern illustrated in Figure 7-a (or that pattern with all orientations reversed).

Since each oriented line in this representation has two different interpretations (a "+" line entering the region associated with the cut-set from that direction or a "-" line entering the cut-set from the opposite direction) there are sixteen different picture cut-sets to examine. One of these is shown in Figure 7-b. A possible associated trace is given in Figure 7-c. The four lines of that trace are of course at right angles to the corresponding picture lines, but their relative lengths are, as yet, undetermined.

The rates of change of w along the lines constituting the trace are equal to the distances from the picture plane origin to the four picture lines. By choosing the origin at the point indicated in Figure 7-b two of the rates of change of w can be made equal to zero. The other two rates of change are then determined and have some fixed ratio. By adjusting the lengths of the segments of trace between A and B and between A and D appropriately it is possible to guarantee that the net change in w around the trace is zero. Consequently there is at least one interpre-

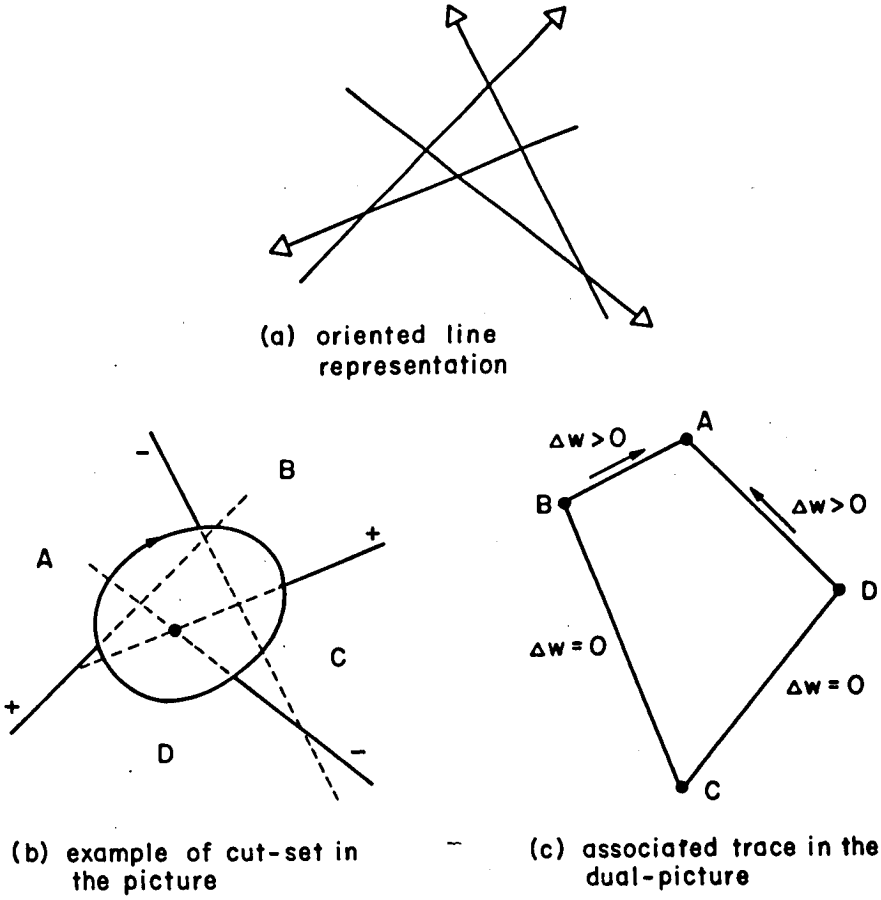


FIG. 7. Example of the basic realizable cut-set of four lines.

tation of the picture lines that demonstrates that the cut-set is realizable. Once we have such an interpretation in mind the location of the origin of the picture plane cannot influence the issue of realizability.

Each of the sixteen cut-sets associated with the oriented line representation of Figure 7-a can also be proved to be realizable in the way illustrated above. Another method for demonstrating realizability is to show explicitly some way of continuing and connecting the cut-set lines inside the picture contour. For our example there are two simple ways of accomplishing this (see Figure 8). Each of the other fifteen possibilities has at least one such simple completion. We have thus completed the second part of our proof for the case of four cut-set lines in general position.

For the special case of a cut-set of three lines that all pass through some common point the proof is even easier, and is left to the reader. The four trihedral configurations of lines that have only "+" or "-" labels in the catalog of

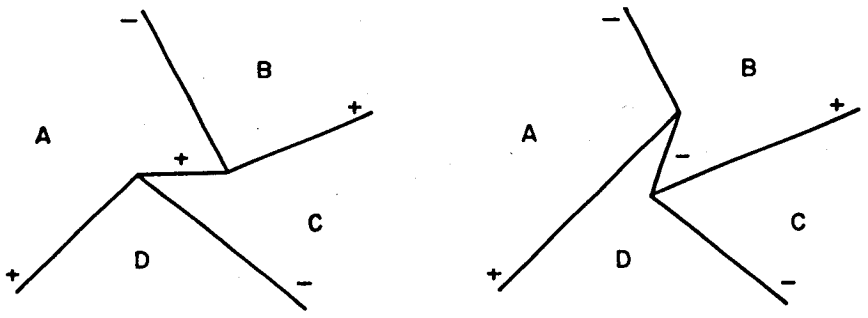


FIG. 8. Examples of completions of the cut-set of Figure 7b.

Figure 1 all correspond to this special case. All are equivalent (as can be seen by the line reversal technique mentioned earlier) to the Y-configuration with all lines labelled "+".

COMPLETION OF THE PROOF

To complete our proof that lack of a ϕ- (or ϕ'-) point implies realizability of a cut-set we need to show that lines can be added to the basic four-line cut-set so that the augmented cut-set is also realizable. We shall give a brief presentation of the essential idea. Consider the realizable cut-set and associated closed trace of Figure 7-b and 7-c. The four components of the trace can be considered to be four vectors that sum to zero and that span the three-dimensional space (u,v,w) of the dual-scene. The addition of some other line to the cut-set in the picture corresponds to adding another vector component to the trace. The direction taken by this new vector component is determined by the orientation of the picture line and its distance from the picture origin. The length of the new component corresponds to the dihedral angle that is associated with the new picture line. If that dihedral angle is zero the new cut-set is obviously realizable.

If that dihedral angle is not zero but has some magnitude that is small compared with the other dihedral angles associated with the original four-line cut-set those angles may be adjusted slightly to compensate for the new line. Although this fact is not obvious when we consider only the picture, it is apparent when we examine the associated dual-scene. Because the original four vector components span the (u,v,w) space the new vector component can be expressed as a positive linear combination of those four vectors. That is, the sum of the new (fifth) vector and a modified positive linear combination of the original four can be made equal to zero. Therefore the trace can be made to close, as it did originally. Addition of other lines to the cut-set can be compensated for in the same way.

A similar argument holds if we add one or more new lines to any one of the basic cut-sets of three lines in the special case in which all these lines are incident at a common point.

We conclude that any cut-set of picture lines (that are labelled with "+" or

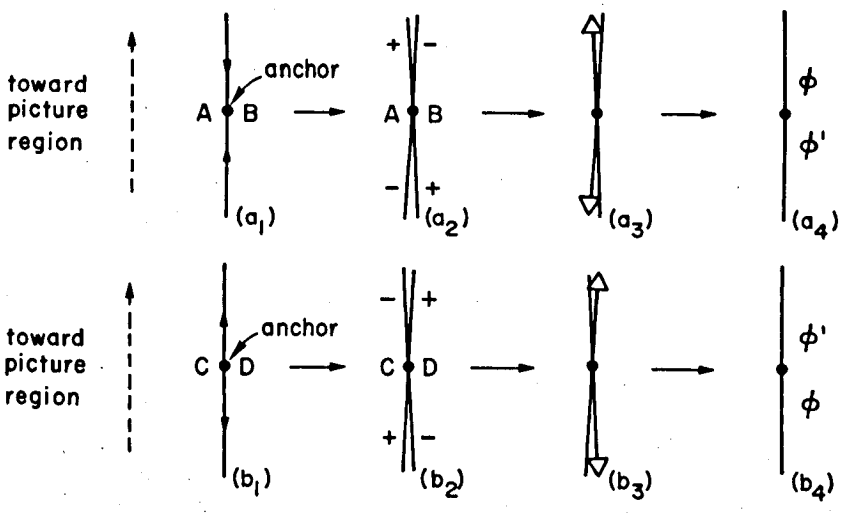


FIG. 9. Picture equivalents for arrow-labelled lines.

"-" labels only) is realizable if and only if the trace can be made to close in the dual-scene. Furthermore, we have seen that this condition is equivalent to the *non*-existence of a ϕ-point or ϕ'-point in the picture plane.

CUT-SETS THAT CONTAIN ARROW-LABELLED LINES

In order to determine the realizability of a cut-set containing arrow-labelled lines we shall replace each such line by a pair of lines, one labelled "+" and the other labelled "-". We recall that an arrow label on a picture line means that the plane to the right of the arrow is closer to the viewer than the region represented to the left of the arrow. This region to the left may be the background against which the other plane is seen or it may simply be a more distant plane surface, perhaps one on another polyhedron.

Consider now the special case depicted in Figure 9-a_1. We consider the two planes A and B, having a common point at the position referred to as the "anchor" at which point the appropriate direction for the arrow label changes. In order for both arrow labels to be appropriate it might be, for example, that the A-plane has a "southward" component of tilt and the B-plane has a "northward" component of tilt. The configuration shown is a most unusual one and would require a special vantage point for the camera, one that would make it appear that the boundary of the A-plane represented above the anchor was an extension of the boundary of the B-plane represented below the anchor.

One physical situation that could be approximated by that representation is that shown in Figure 9-a_2 in which we assume that the angle between the two picture lines is arbitrarily small. The two small sectors could then be thought of as "cliff faces" seen nearly edgewise. If we assume that the direction to the picture region associated with the cut-set is up, as indicated in the figure, the

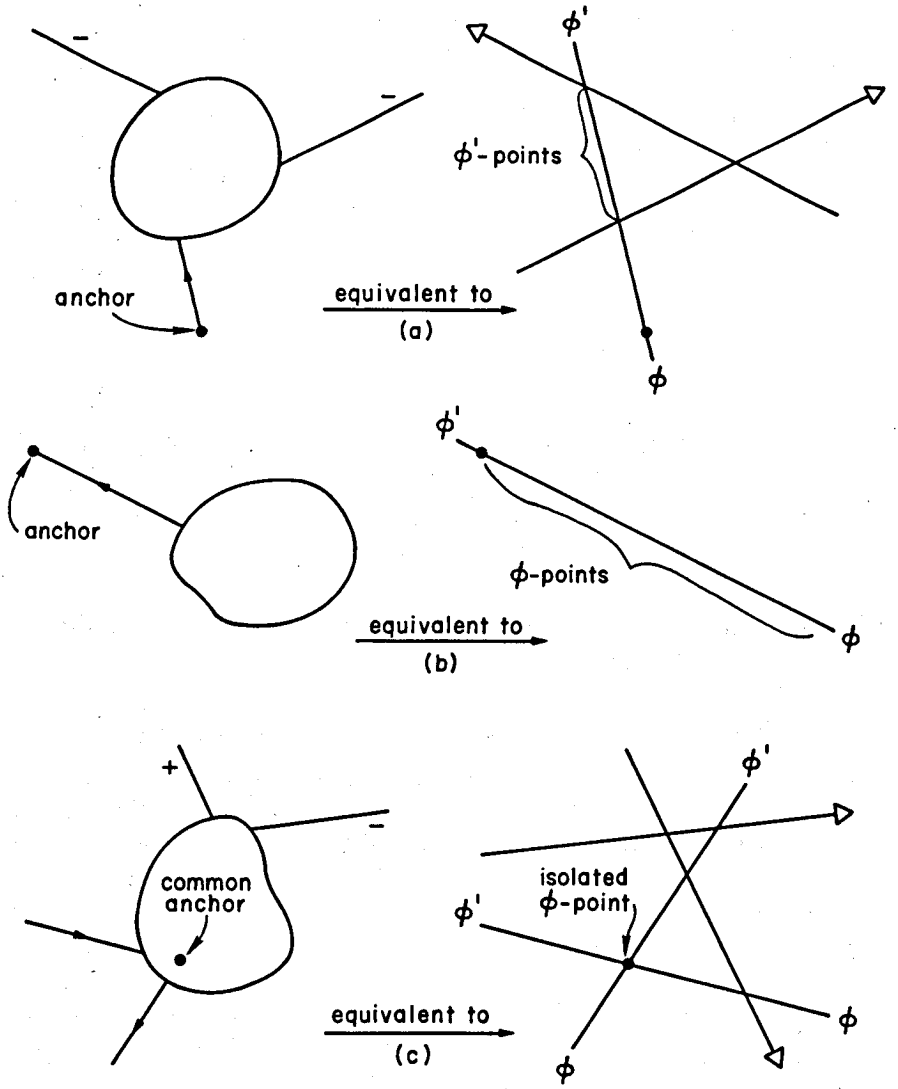


FIG. 10. Three examples of unrealizable cut-sets.

corresponding oriented line representation would be that of Figure 9-a_3. In the limit (as the angle between the two lines approaches zero) we see that the only possible location for ϕ-points is on the line above the anchor; the only possible location for ϕ'-points is on the line below the anchor. The equivalent representation in Figure 9-a_4 indicates those facts. Another special case, shown in Figure 9-b, establishes an equivalent for a related situation in which the plane C is tilted north and the plane D is tilted south.

In order to determine the realizability of a cut-set containing one or more arrow-labelled lines each such line is replaced by its equivalent (either from

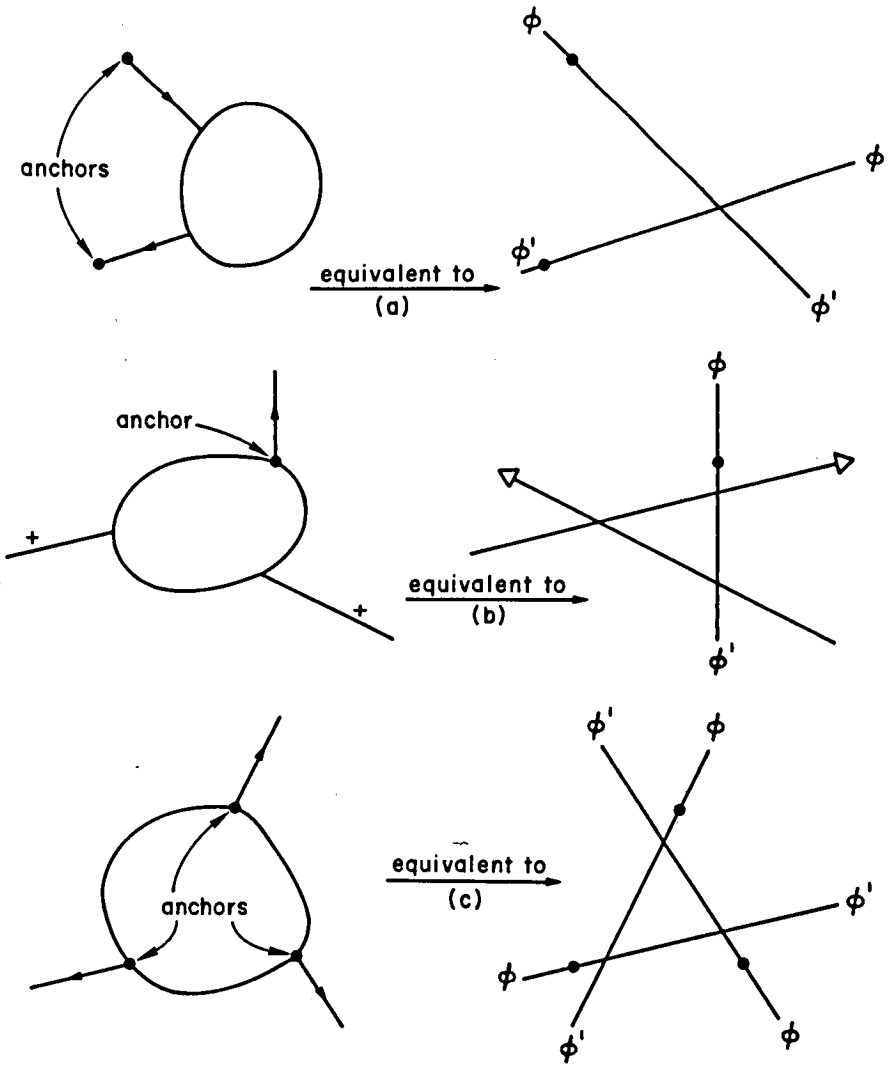


FIG. 11. Three examples of realizable cut-sets.

Figure 9-a_4 or 9-b_4). Any possible ϕ- (or ϕ'-) point must lie *on* the corresponding half-lines of all such equivalents as well as being to the right of (or the left of) all oriented equivalents to "+" or "-" labelled lines. Figures 10 and 11 give some simple examples of both realizable and unrealizable cut-sets.

In general, the position of the "anchor" on an arrow-labelled line may have to be known before the realizability of the cut-set containing that line can be determined. If the region to the left of an arrow-label is an infinitely remote background the placement of the anchor on the line may be chosen arbitrarily. In that case, however, the locations of possible ϕ- (or ϕ'-) points for the picture are still constrained to be only on that line.

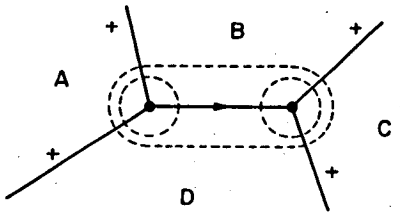


FIG. 12. A doubly-anchored arrow-labelled line.

Special cases of realizable cut-sets that contain arrow-labelled lines are included in the catalog of Figure 1. It is easy to prove that any single-node cut-set that contains one of the configurations of that catalog is also realizable.

A special case worthy of attention is that of a configuration of lines that constitutes a "T". In the restricted trihedral language (Huffman, 1971) this configuration can only represent one scene edge obscuring another. Thus in that language the bar of the "T" should always be assigned a left-pointing arrow-label. The node at the junction of the three line segments does not correspond to a vertex of a polyhedron in that case.

When arbitrary numbers of planes can be incident at a common polyhedral vertex a "T" can have many other interpretations. This configuration requires special treatment because two of the components of the configuration lie along exactly the same line. In this paper there is included in the Appendix a catalog and brief summary of results pertaining to the interpretations that a "T" may have.

GENERAL CONDITIONS FOR REALIZABILITY OF A PICTURE

It would be tempting to speculate that if all cut-sets of a picture are realizable then the picture itself is. That unfortunately is not the case. Consider, for instance, the portion of picture shown in Figure 12. The cut-sets associated with both the left and right picture nodes are each easily proved to be realizable (each contains a subset of labelled lines contained in the catalog of Figure 1). It is also easy to prove that the larger cut-set assoicated with the pair of nodes is realizable (it is an example of the basic four-line cut-set discussed earlier). By examining the left node we can see that the planes A, B, and D would all have the associated vertex in common. Similarly we see that the planes B, C, and D all would contain the other vertex. Thus there is evidence that the arrow-labelled line is anchored at each of those two points. This is, of course, ridiculous. The plane D cannot be nearer the viewer along the line than plane B is and at the same time have two points in common with plane B.

Another more complicated example of a portion of picture that is not realizable is shown in Figure 13. There are seven ways of enclosing subsets of the three nodes and thus seven cut-sets, each of which is itself realizable. The one associated with all three nodes is effectively the same as in the third example in Figure 11. The others are left as exercises for the reader.

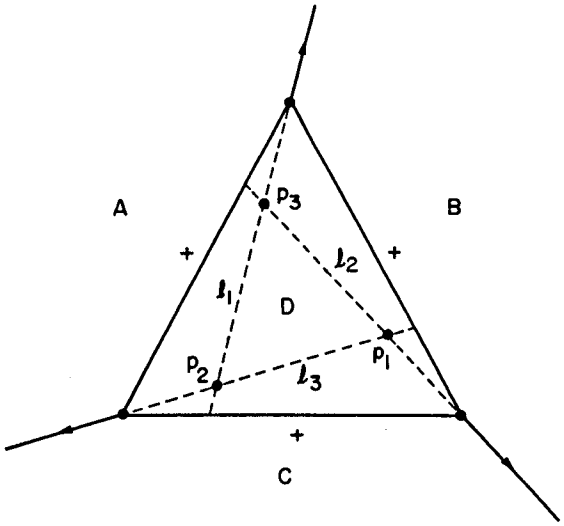


FIG. 13. An unrealizable portion of a picture that has all cut-sets realizable.

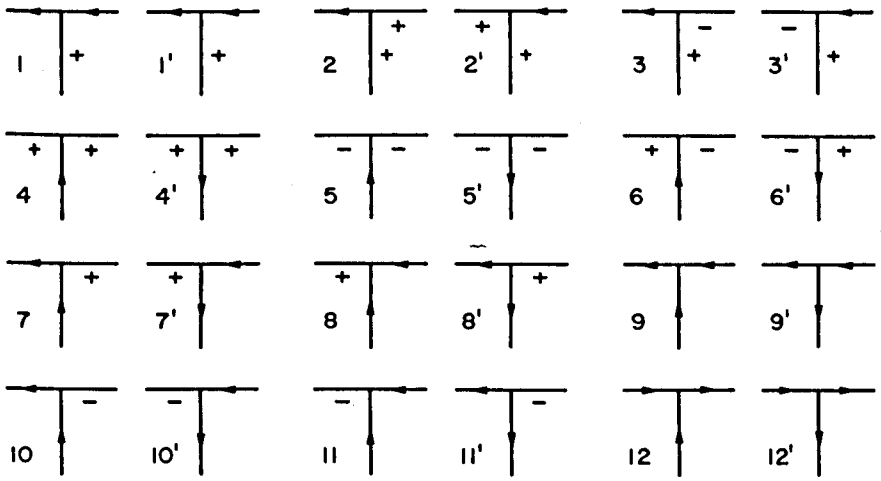


FIG. 14. Realizable T-configurations requiring no special vantage point.

The fact that the picture is nevertheless not realizable can be demonstrated as follows. We note that the anchor (the top picture node) on the arrow-labelled line implies that on the extension of this line on the other side of the anchor the B-plane would be further away from the viewer than would the A-plane. Thus along the segment ℓ_1 we have $z_A < z_B$. Similarly, along ℓ_2 we have $z_B < z_C$ and along ℓ_3 we have $z_C < z_A$. Therefore at p_1 (the point common to ℓ_2 and ℓ_3) we have $z_B < z_A$. Therefore somewhere between p_1 and ℓ_1 is the locus (a straight line) along which $z_A = z_B$. Similarly between p_2 and ℓ_2 is a line along which $z_B = z_C$ and between p_3 and ℓ_3 is a line along which $z_C = z_A$. We conclude that

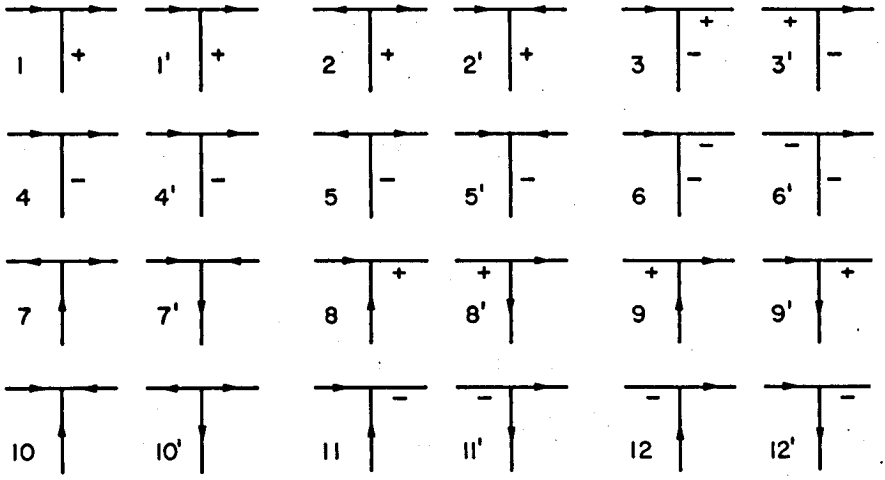


FIG. 15. Realizable T-configurations requiring a special vantage point.

somewhere inside the triangle bounded by the dotted lines is a picture point p_o at which $z_A = z_B = z_C$.

Imagine now a line drawn between p_o and the top picture node. That construction line would obviously have to lie to the left of (counterclockwise from) the arrow-labelled line through that top node. Along that construction line the A- and B-planes if extended would intersect and that line of intersection would be convex (be associated with a "+" label) as seen from the vantage point of the viewer. We would, in turn, conclude that the region in which the topmost arrow-labelled line lies is one for which $z_A < z_B$. But this conclusion is in contradiction with $z_A > z_B$ which is implied by the arrow label on the line itself. The contradiction is apparent and we conclude that the picture is not realizable.

SUMMARY

We conclude that the realizability of all cut-sets of a picture, the lines of which have been tentatively labelled, is a necessary but not sufficient condition for the realizability of the picture with those "meanings" associated with the lines. The tests for realizability developed in this paper nevertheless are easy to apply to the various cut-sets and do, in effect, generalize as far as is possible the catalog of trihedral configurations (see Figure 1) derived earlier (Huffman, 1971).

It seems likely to the author that labelled pictures that pass the cut-set tests developed in this paper but are still not realizable (see, for instance, Figures 12 and 13) will, as a practical matter, have to be tested for realizability by simply attempting to find a three-dimensional dual representation (see Huffman, 1976) consistent with the picture. This is, of course, an exceptionally weak claim since it is equivalent to saying that a picture is realizable if and only if it is possible to determine locations of planes in the scene that intersect and obscure each other

in ways that yield the lines of the given picture.

APPENDIX: CATALOG OF REALIZABLE T-CONFIGURATIONS

In a polyhedral picture a "T" is an example of a configuration in which two or more lines have interdependent directions. In the case of a "T" the pair of lines constituting the top crossbar have exactly the same direction. Perhaps the simplest interpretation of a "T" is that the crossbar depicts an edge that obscures the edge represented by the third line. In that case the junction of the "T" does not represent a vertex of the scene. If that junction does represent a polyhedral vertex than many other interpretations are possible, all requiring more than three planes at that vertex.

Since there are four possible line labels there are $4^3 = 64$ ways of labelling the three lines of a "T". Nineteen of these are not realizable. Of the remaining 45 that are realizable there are 23 (see Figure 14) that have the property that the two components of the crossbar represent a single line in the three-dimensional scene. These 23 configurations thus do not change their basic form when photographed from slightly different camera positions. We note that the "-" label is never appropriate on the third line of the "T". The remaining 22 labelled configurations are realizable but would require a special vantage point for the camera in order for the picture to show the two crossbar components exactly aligned.

The entries of Figures 14 and 15 are arranged in pairs each member of which is a mirror-image of the other. When the mirror image of a scene is labelled one should keep the "+" and "-" labels of the resulting picture the same. The directions of arrow-labels on the remaining lines should be these obtained by reversing the directions on the arrow-labels when the original line-labelled picture is viewed in a mirror. Some labelled configurations (No. 1 in Figure 14 and No. 1 and No. 4 in Figure 15) remain invariant under this transformation.

Three of the catalog entries (No. 12 and No. 12′ in Figure 14 and No. 1 in Figure 15) have the additional property that they are not realizable unless the 180° sector above the crossbar represents a distant "background". That sector cannot represent a plane that contains the indicated vertex as one of its points.

REFERENCES

Huffman, D.A. (1971) Impossible objects as nonsense sentences. *Machine Intelligence* 6 (eds. Meltzer, B., and Michie, D.) Edinburgh University Press, Edinburgh, pp. 295-323.

Huffman, D.A. (1976) A duality concept for the analysis of polyhedral scenes. This volume.

25

How to See a Simple World: An Exegesis of Some Computer Programs for Scene Analysis

Alan K. Mackworth

Department of Computer Science
University of British Columbia

This is not a comprehensive survey of machine vision which, in its broadest sense, includes all computer programs that process pictures. Restricting attention to scene analysis programs that interpret line data as polyhedral scenes makes it possible to examine those programs in depth, comment on revealing mistakes, explore the interrelationships and exhibit the thematic development of the field. Starting with Roberts' seminal work which established the paradigm, there has been an evolutionary succession of programs and proposals each approaching the problem with a different emphasis. In addition to Roberts' program this paper expounds in detail work done by Guzman, Falk, Huffman, Clowes, Mackworth, and Waltz. These programs are presented, compared, contrasted and, sometimes, criticized in order to exhibit the development of a variety of themes including the representation of the picture-formation process, segmentation, support, occlusion, lighting, the scene description, picture cues and models of the world.

PREAMBLE

As this paper focusses on polyhedral scene analysis, it should not be read as a review of all recent work in computational vision. The semantics of polyhedral scenes are so clean that we can review that body of work and see it as a coherent whole. On the other hand much recent work outside that area is so diverse and fragmented in character that it is hard to place it all in a single framework. However, the associated lecture will cover such topics as the interpretation of more complex scenes and the question of how image analysis (for example, line and region formation) can be guided by partial scene analysis. Within the area covered here the major omission is the MIT COPY DEMO which is so ably described by Winston (Winston, 1973).

Caveat lector: one of the techniques used in this review is to point to non-trivial bugs in the programs discussed. These are useful for gaining insight into the weaknesses of the descriptions and inference mechanisms available to a program; however, it must be emphasised that, for the most part, these have been discovered not through running the program in question but through a

careful reading of the published accounts. To seek refuge in the fact that most of these bugs could be fixed by admittedly *ad hoc* patches would be to mistake the symptoms for the disease.

INTRODUCTION

The Platonic assumption that the world is made up entirely of objects with flat surfaces obviously does not hold; and yet, as with so many other simplifications of reality for the sake of tractability, it has been immensely productive in establishing a paradigm for scene analysis. There is a coherent evolving body of research based on the notion that a polyhedral world is the simplest we can consider without eliminating any of the essential aspects of scene analysis, namely, the picture-taking process, models, lighting, support, occlusion, and so on. The thesis is that once we achieve ways of dealing intelligently with those aspects for a simple, but nonetheless real, world we could then consider the fuzzy world of teddy bears (Michie, 1974) and the like. This should not be taken as suggesting that each of those aspects presents simply a separate, independent subproblem to be solved. The most important question to be faced was how to write programs that coordinate the use of these separate, but interrelated, knowledge systems to achieve sensible picture interpretations. Roberts (Roberts, 1965) was the first to give an answer to this question. We shall examine his answer in some detail, because he exposed in it the issues that became themes of the first decade of scene analysis.

ROBERTS' PROGRAM FOR SCENE ANALYSIS

Roberts (Roberts, 1965) described a program for the interpretation of photographs as images of fully three-dimensional scenes. By assuming that the scene is composed of particular instances of object models that have been transformed and combined in well-specified ways and by using knowledge of the picture taking process, support and occlusion, his system is able to compute the exact 3D position of every object in the scene. There are actually two separate programs. The first reduces the photograph to a line drawing, the second interprets the line drawing. The reduction to a line drawing does not concern us here because an adequate treatment of that topic is beyond the scope of this paper and because more recent work on line finding (Shirai, 1973; O'Gorman and Clowes, 1973) suggests that the simple, pass-oriented line-following procedures Roberts describes are not usually powerful enough to produce the complete line drawing required by the subsequent interpretation program.

Roberts' program believes that the world consists of the models shown in Figure 1, namely, a cube, a rectangular wedge and a hexagonal prism. To create simple objects the system allows these models to be expanded along each of the model coordinate axes and then rotated and translated. Compound objects are created by abutting two or more simple objects so that each adjacent pair shares a common surface. The models are specified by 3D homogeneous coordinates so

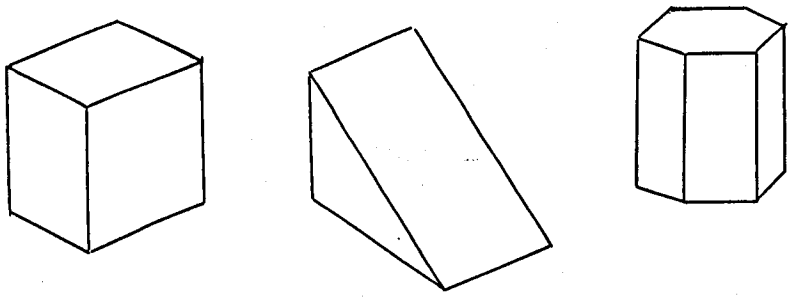


FIG. 1. Roberts' simple object models

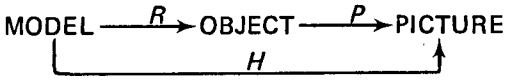


FIG. 2. Roberts' domains and transformations

that the transformation of a model to form an object is described as the transformation, by an initially unknown matrix R, of the coordinates of the corners and the normals to the surfaces. Similarly the perspective picture taking process is described as the multiplication by a known matrix P of the object coordinates to produce the picture coordinates followed by the removal of hidden lines. So the relationships of the model, object and picture domains are as shown in Figure 2 where H, the model-to-picture transformation, is also shown. Since $H = RP$, if a model and a transformation H can be found that account for a set of the lines in the picture then the program maintains that the set of lines is a picture of the object given by a transformation $R = HP^{-1}$ of that model. Thus the object is identified and its location specified completely except for its actual distance from the camera. This distance is then computed from the requirement that the most downward facing surface of the object must lie in the ground plane. This is the only support hypothesis used by the program.

In this abbreviated account the most important point that has been glossed over is the decision to choose a set of picture lines to account for. This decision is followed by the choice of particular edges of a particular model to account for those lines. This is perhaps the archetypal artificial intelligence problem—the problem of relevance, by which is meant the problem of invocation of appropriately relevant models or procedures to account for the data.

The space of three models juxtaposed and transformed in all possible ways and viewed from every direction is unthinkably large for a blind search, (that is, generating all possible pictures of all possible objects until one matches the input) so the search space must be intelligently structured. Roberts noticed that all the model transformations leave the object's topology invariant and that within a wide range of viewpoints the topology of the visible aspect of an object does not change. Through this invariance the topology of the picture can be used to search a much reduced space consisting of the models viewed from a small

number of typical viewpoints. On finding a candidate model, points that correspond in the model and the picture are paired. The coordinates of those pairs are used to calculate (rather than search for) the model-to-picture transformation, H. At least four pairs of points are needed to calculate H; if more are available then a least squares fit gives H with the residual error as a measure of the picture-model mismatch. If the mismatch is too large then that model is rejected and the topology search continues.

Consider the topology search in detail. It is based on the notion of an approved polygon which is simply one of the shapes of the model surfaces. For the three models used, an approved polygon is any convex polygon of 3, 4 or 6 sides. Since the topology search attempts to find the largest picture fragment that could correspond to a model, it proceeds in stages each of which looks for a smaller fragment than the one preceding. The four stages, which are called in sequence until one succeeds, are:

1. Find a picture vertex surrounded by 3 approved polygons.
2. Find a line with an approved polygon on each side.
3. Find an approved polygon with an extra line coming from one vertex.
4. As a last resort find a point with 3 lines coming from it.

When a suitable fragment is found the program searches the models in sequence (cube followed by wedge followed by prism) to find a topological structure that corresponds to the fragment recovered from the picture.

Figure 3 (a) shows a typical compound object considered by Roberts. The topology search finds no fragments of type 1, but two of type 2: both lines 2 and 3 have approved polygons on each side of them. The cube has quadrilaterals on both sides of an edge so the geometry matcher tries A and B as surfaces of a transformed cube as shown in Figure 4, but discovers that the residual error of the least squares fit of the corresponding object-model point pairs is too large and rejects it. Similarly for line 3. The topology search then turns up a type 3 fragment: polygon A with line 9 attached. The five points defined by that fragment match a transformed cube exactly as in Figure 3 (b). This is removed from the original picture and the process continues by finding the parts shown in Figure 3 (c) and (d) with the final compound object shown in Figure 3 (e).

There are some very real difficulties with this program which can be illustrated by considering specific cases. In the example above, take the rejection of a cube model for surfaces A and B across line 2. Certainly if the projection is without perspective so that lines 1, 2, and 3 are parallel as are 5 and 6, 7 and 8 then a transformed cube fits easily as the rectangular solid in Figure 4 shows. This would be disastrous for the subsequent analysis. Thus Roberts' claim (Roberts, 1965, p. 166) that "the process accounts for but does not depend on perspective information" seems to be wrong. In the perspective case the convergence of lines 5 and 6 can be used to reject it. Even assuming that the line fitting is so accurate that such fine distinctions can be made reliably, doubts must be

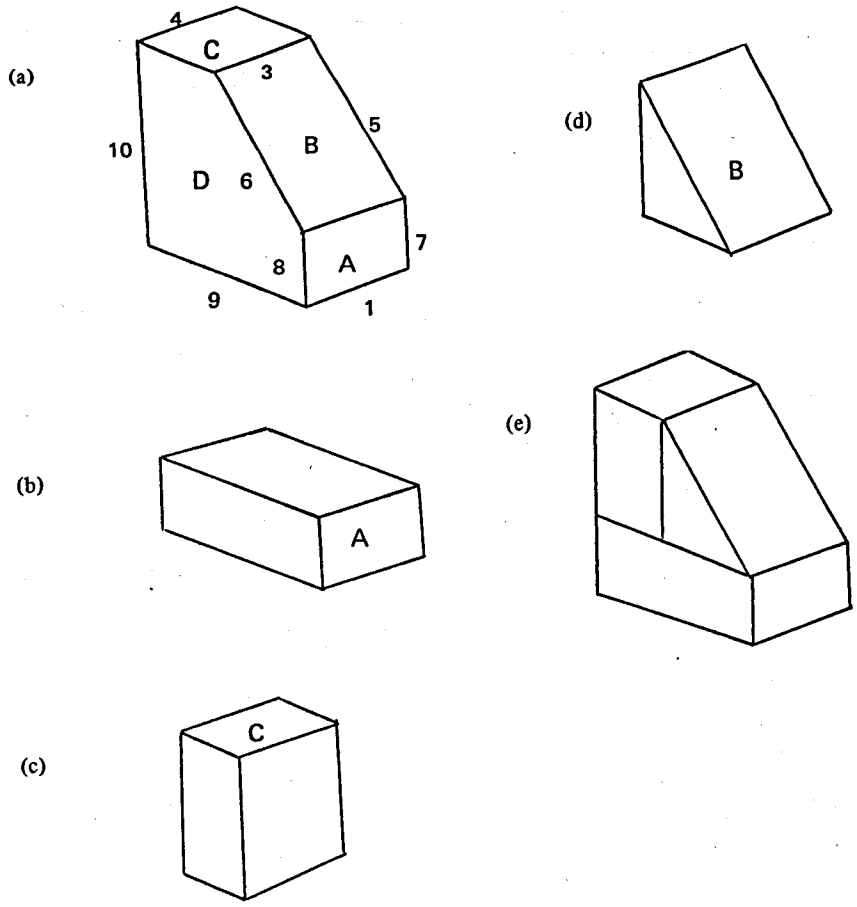


FIG. 3. Interpreting a compound object

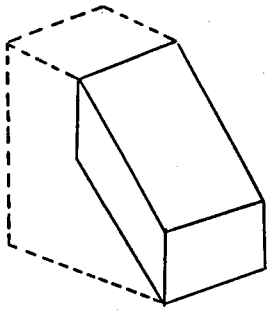


FIG. 4. Seeing a transformed cube in a compound object

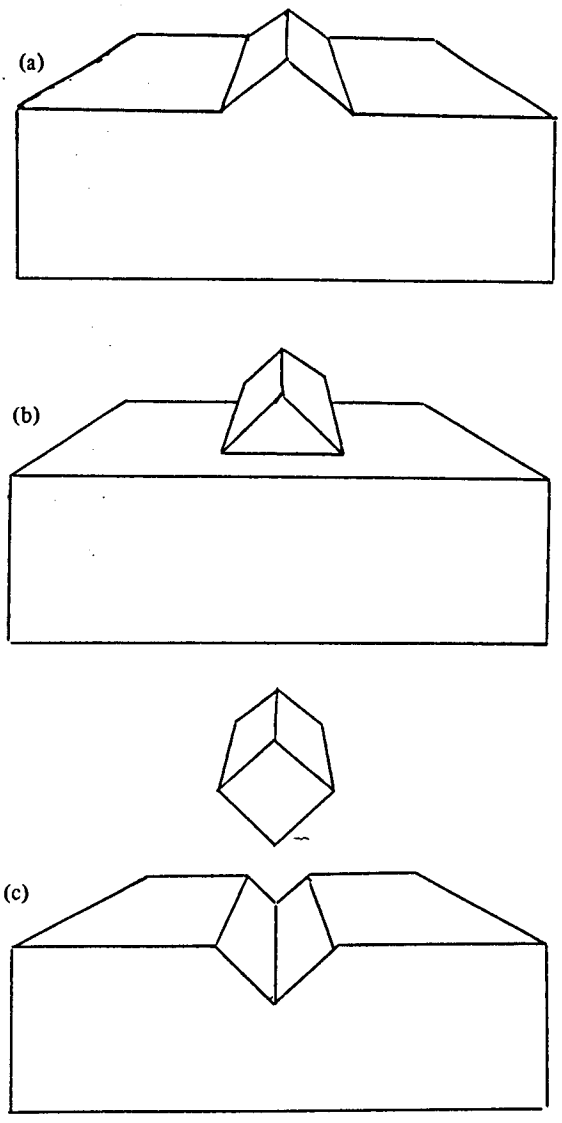


FIG. 5. Two decompositions of an object

raised about a system that depends on such distinctions.

Another example is the compound object of Figure 5 (a). Given the three basic models the program could be expected to split it into the two simple objects of Figure 5 (b). But in fact it will first remove a cuboid from the top surface as in Figure 5 (c) which leads into a muddle because it has not taken the appropriate first step. This arises because the models are tested in strict sequence: cube, wedge, prism. That ordering is used to avoid splitting a cube into two wedges!

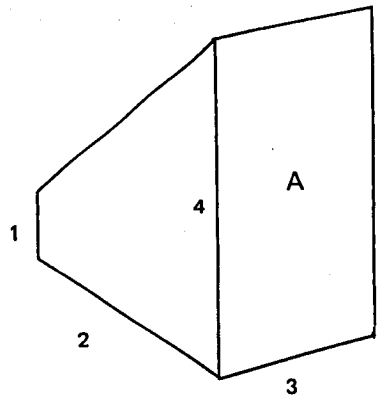


FIG. 6. Another compound object

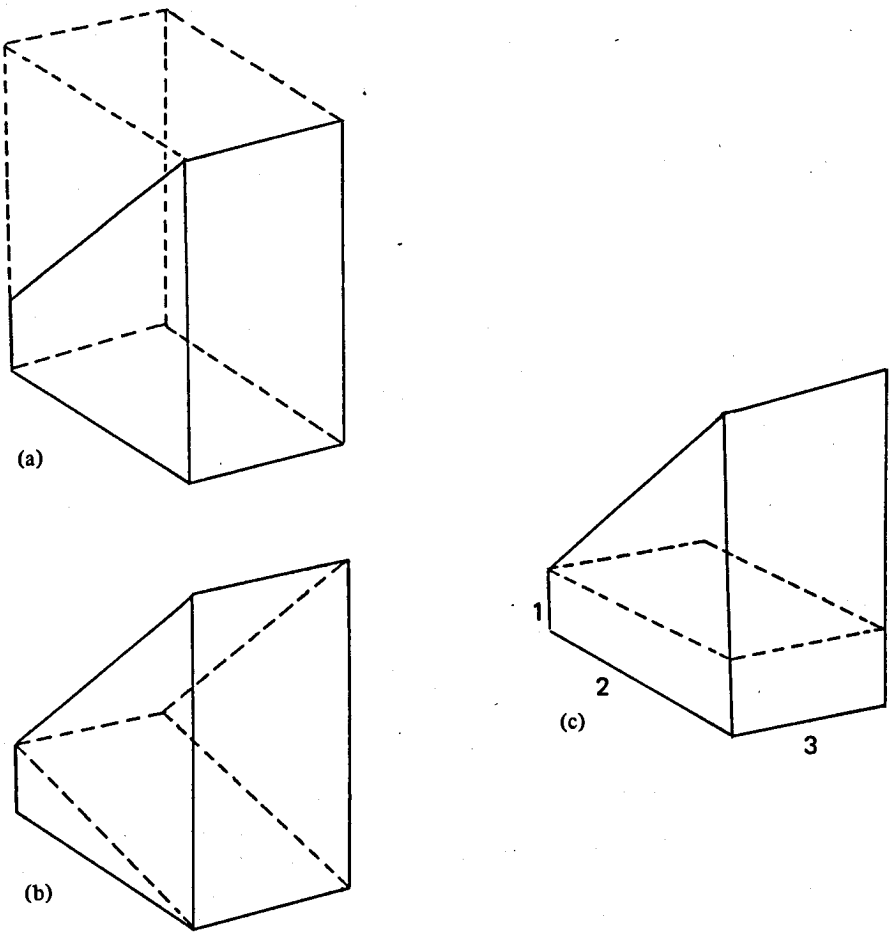


FIG. 7. Possible decompositions of the object in Figure 6

Finally consider the simple picture in Figure 6. This object is simply a wedge on top of a cuboid. But as the program is followed through on this picture it appears that whenever the topology tests succeed the model suggested will not pass the geometric transformation test, and so the program fails completely.

The topology test finds the two quadrilaterals flanking line 4 but if one face of the cube is fitted to region A the rest of the cube will fall outside the complete figure as Figure 7 (a) shows. Attempts to fit wedges or cubes using quadrilaterals with an extra line from one corner will all fail. In particular Figure 7 (b) shows a wedge that might be thought to fit but it is incorrect as only rectangular wedges are allowed. Finally, even withdrawing to just three lines from a vertex will not succeed. Looking at lines 1, 2, and 3 of Figure 7 (c) they can be seen to be three significant edges of a cube model that could be made to fit but the program does not find that context as it only looks for contexts concentrated at vertices. Finin (Winston, 1973) has defined the skeleton of a cuboid to include the sort of context needed here.

Despite the difficulties uncovered above, Roberts' program created a scene analysis paradigm that remains dominant. As a working theory, for that is what an AI program is, it firmly established an active model of perception as a cycle of four processes: discovering cues, activating a hypothesis, testing the hypothesis, and inferring the consequences. This model of perception, so far removed from the then dominant pattern recognition paradigm for machine perception, echoes, as Clowes (Clowes, 1972) remarked, the approach of such psychologists as Helmholtz (Southall, 1962), Bartlett (Bartlett, 1967), and Gregory (Gregory, 1974). Minsky's frame systems (Minsky, 1975) provide a semi-formalism for this paradigm of perception.

GUZMAN'S BODY SEGMENTATION PROGRAM, SEE

Guzman's SEE (Guzman, 1968) accepts line diagrams of polyhedral scenes as input and partitions the picture regions on the basis of the putative body membership of the surfaces depicted. The program consists of two passes over the picture. The first pass makes local guesses (called links) about which pairs of regions depict the same body. The second pass accumulates that evidence to produce a grouping of the regions corresponding to bodies.

The links are placed at the junctions shown in Figure 8 where the links are shown as connections between two regions which are usually adjacent in the picture. An exception to these rules is the inhibition rule that no link is placed across a line at a junction if its other end is a barb of an ARROW, a leg of an L or part of the cross bar of a T.

Considering the result of the first pass to be a graph with regions as nodes and links as arcs then the second pass searches for 2-connected subgraphs which are declared to represent bodies. This is a highly abbreviated version of Guzman's final account which has many special case rules augmenting both passes. The rules that depend on being told which region is background can clearly be invalidated immediately by putting another block behind the scene being ana-

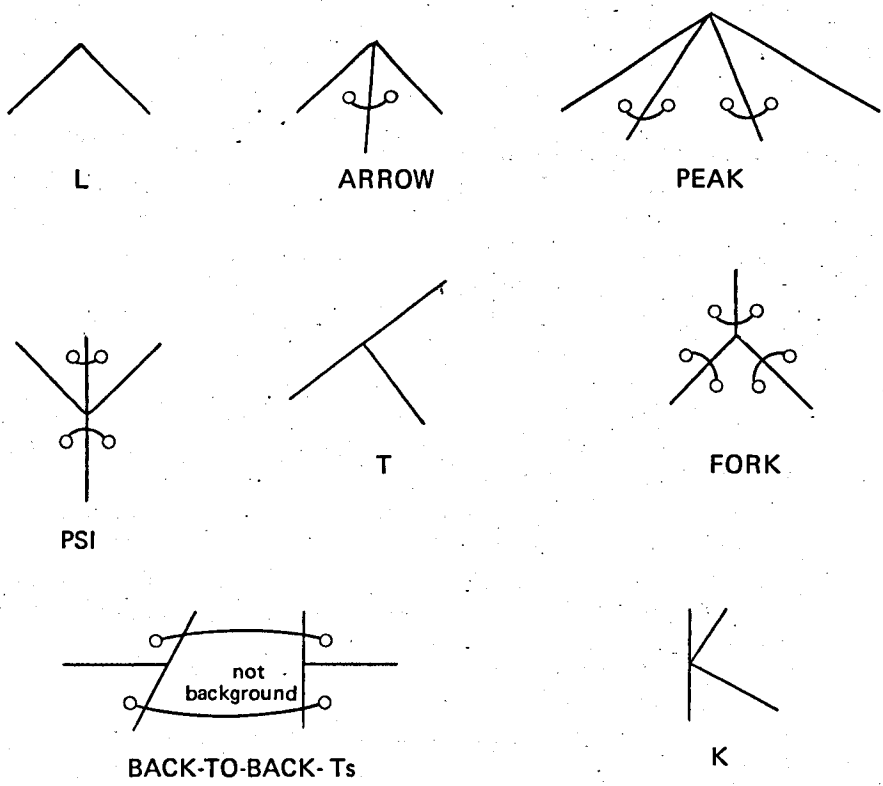


FIG. 8. The junction categories and link planting rules of SEE

lyzed. That, however, is not the main point; it is merely typical of the way in which the program developed by a process of finding counter-examples that both invalidated old rules and hinted at new ones (Winston, 1973). The need to add and modify rules almost continuously to handle exceptions suggests that there is a basic flaw in the design.

The flaw seems to be that Guzman used locally computed picture predicates as evidence for global scene-based properties. To avoid this one must ask what do the lines in the picture depict? As we shall see later in the Huffman-Clowes labelling algorithm they can depict many things but only certain combinations of these things are scene coherent; this coherence decision cannot be made in the picture domain as Guzman tried to do.

SEE's tendency to see holes in objects as separate objects (Winston, 1968) is only one consequence of the fact that the program ignores ambiguities inherent in the interpretation process that are exposed by the Huffman-Clowes labelling algorithm. For example, consider Figure 9 (a) [adapted from (Minsky and Papert, 1972)]. That can be seen in at least three different ways. The first possibility is as a simple house structure in which there is only one body. Second, as a variant of the first it can be seen as a pyramid sitting on top of a

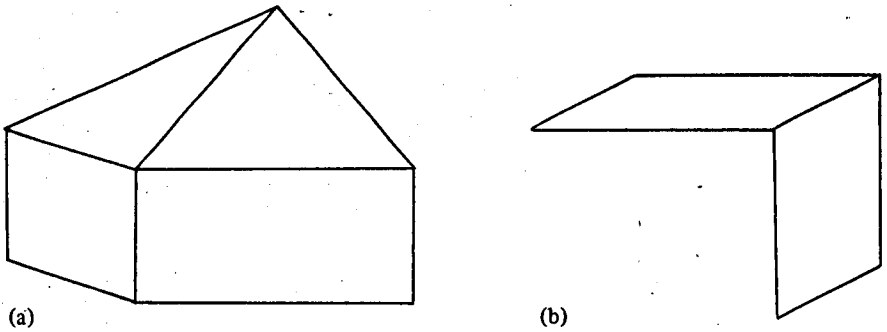


FIG. 9. Illustrating ambiguity (a) and anomaly (b) for SEE

rectangular brick. Third, and quite different from the first two, it could simply be two wedges abutting one another. SEE reports only the first of these alternatives and does not see the others. Moreover, SEE's interpretation consists only of "one body composed of regions A, B, C, and D;" it does not provide the richness of an interpretation that reports the nature of each edge. These ambiguities and that richness are provided by the labelling algorithm (Waltz' version is needed for Figure 9 (a)) as we shall see. The labelling algorithm also detects situations illustrated by the picture in Figure 9 (b) where SEE happily partitions into bodies pictures that are syntactically correct (that is, every line bounds two different regions and so on) but meaningless as pictures of polyhedra.

An interesting comparison can be made between SEE and Roberts' program. Roberts initially hopes to find a picture fragment that corresponds to a part of one of his three prototypes so that the regions offered up should at least belong to the same body. Recalling that an acceptable polygon must be a convex region, if the first stage of the topology matching succeeds (3 acceptable polygons around a vertex) then it will return a FORK vertex with all three regions hopefully depicting surfaces of one body. This corresponds directly to the most powerful Guzman heuristic–the FORK that plants three links. If the first stage of Roberts' topology matching fails and the second stage (2 acceptable polygons flanking a line) succeeds then that line is almost certainly the shaft of at least one ARROW, so the second stage of Roberts' topology matching corresponds to the second most powerful Guzman heuristic linking the two regions flanking the shaft of an ARROW. Furthermore, in both the above cases, Guzman's inhibition of a link across a line at a junction if the other end of that line is a barb of an ARROW or a leg of an L corresponds directly to the convex region requirement of Roberts.

This comparison could easily be continued (consider the corresponding uses of T-junctions) but it has gone far enough to make three points beyond observing the intriguing parallels. In the first place it is now obvious that Guzman's work is not as radically new as it appeared to be. In the light of the analysis, Waltz' (Waltz, 1972) claim that "indeed his approach was a dramatic departure

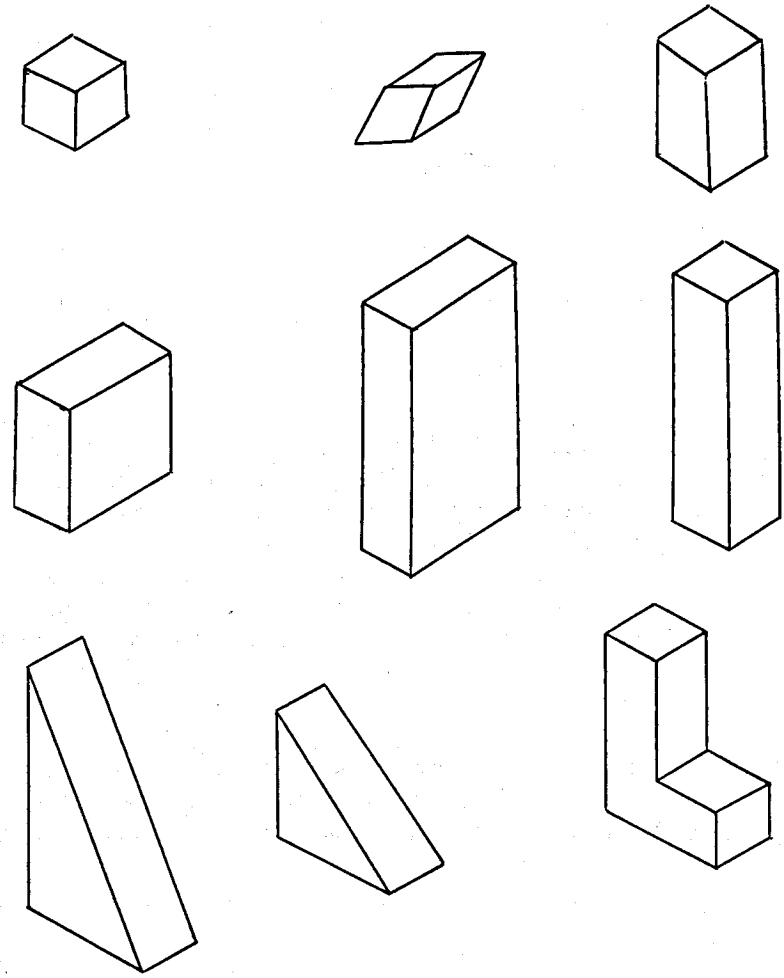


FIG. 10. The object prototypes of INTERPRET

from what had been done before him" appears to be over-enthusiastic. Second, we notice that Guzman did not even use such simple properties of regions as 'convex' but instead tried to express such a slightly less locally confined picture property in terms of his complicated inhibition rule based entirely on junction geometry. Third and far more important, Roberts used knowledge of prototypes explicitly in the body segmentation problem. He did this in three ways, first by using a general property (acceptable polygon) of all the prototypes, and prototype-specific topology tests to identify a picture fragment as part of a prototype and then, having made an identification, projecting the rest of the prototype onto the picture to account for many more lines. Guzman on the other hand claims to use no knowledge of prototypes in the segmentation. This claim may indeed be doubted on the ground of the Roberts-Guzman parallel presented

FIG. 11. The organization of INTERPRET

here. SEE seems to prefer convex regions as body faces. This is confirmed in the analysis of SEE's underpinnings below. This claim to virtue (as it was seen by Guzman) in fact turned out to be an objection to SEE as it led to a vision system that was pass-structured with successive passes mapping into progressively more abstract domains (Minsky and Papert, 1972).

FALK'S SCENE ANALYSIS SYSTEM: INTERPRET

Falk's (Falk, 1972) collection of scene analysis programs operating as a system called INTERPRET represents a gathering together of the state of the art in scene analysis *circa* 1970. Given a range of nine fixed size prototypes that appear in the world (Fig. 10) and the position and orientation of the ground plane relative to the picture plane, the system is required to interpret line drawings (with, possibly, a small number of lines missing) to produce an exact 3D representation of the scene.

The system consists of the five stages of Figure 11. SEGMENT partitions the set of picture lines into bodies. For each body, SUPPORT determines the set of bodies that could conceivably support it. COMPLETE tries to add lines to the picture of each object so that RECOGNIZE will find it easier to identify it as one of the prototypes. RECOGNIZE also determines the position of the prototypes so that PREDICT can say what the picture should look like. Finally VERIFY determines if the predicted and given picture match. The system is strictly pass structured with the five stages called in sequence with the exception that a failure in VERIFY requires RECOGNIZE to produce another suggestion.

SEGMENT used Guzman-type vertex classifications to assign edges to bodies. It assigns edges rather than regions as SEE did because the possibility of edges not being depicted means that a single region could correspond to two surfaces of separate bodies. Each Guzman vertex category is split into two: GOOD<category name>and BAD<category name> on the basis of local context that can include adjacent junctions. The hope is that, for the most part, GOOD junctions show edges of only one body while BAD junctions show edges of more than one body. As an example of the GOOD/BAD distinction, an ARROW is a BADARROW if one of the regions flanking the shaft is background or if the shaft is the top of a K junction, otherwise it is a GOODARROW. The next step determines sets of lines such that each set connects a group of GOOD vertices. Each set then represents edges of a single body. The total set of lines thereby assigned does not necessarily exhaust the set of lines in the picture. SEGMENT then assigns regions to bodies based on the line segmentation and a few extra heuristics for splitting regions that correspond to more than one body.

RECOGNIZE needs to know which bodies in the scene could support other

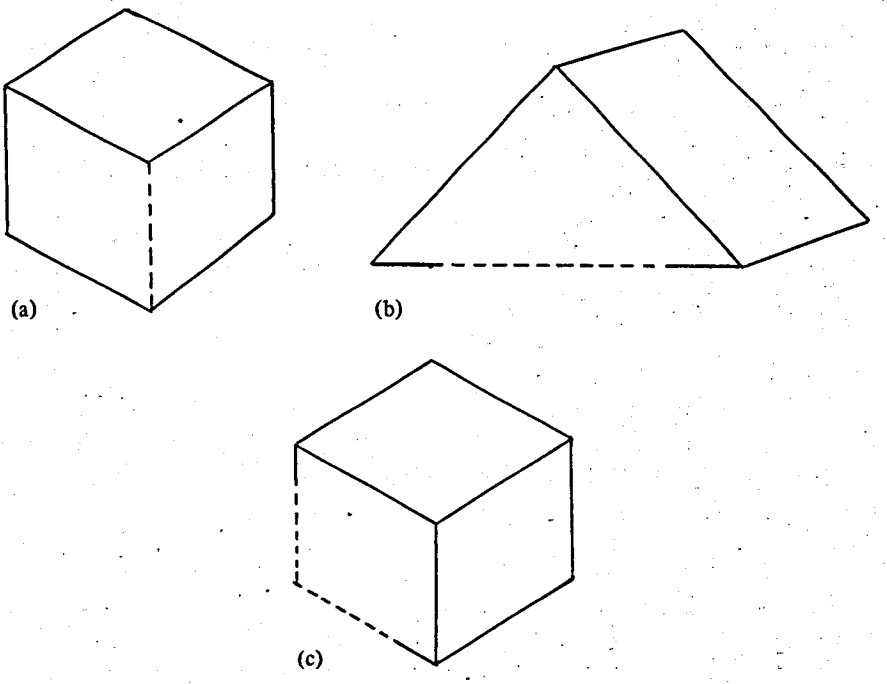


FIG. 12. The contexts for COMPLETE

bodies because it infers the position of each body from the position of the body supporting it, that is, working up from the known position of the table. SUPPORT creates the set of potential supporters for each body. It starts by establishing which are the base edges of each body by applying six elimination filters to the set of exterior lines for each object. For example, eliminate both lines at downward open L vertices. These filters all depend on the local picture geometry of each line. SUPPORT then defines the potential supporters for the body as those bodies that have a face appearing adjacent to one of the base edges. If a body has only one potential supporter then that must be the actual supporter. In particular for objects supported by the background surface, RECOGNIZE will be able to establish the 3D position of the endpoints of all the base edges.

The picture of each object may be incomplete for three possible reasons: (a) the original picture had some lines missing or (b) the object is partially occluded or (c) SEGMENT failed to assign some lines to the body. COMPLETE has three routines that attempt to patch up each object before recognition. Figure 12 shows dotted lines where ADDLINE, JOIN and ADDCORNER fill in lines. ADDLINE seems intended for case (a), JOIN and ADDCORNER for case (b). ADDLINE puts a line between two L vertices that open upwards and have parallel arms.

INTERPRET does not recognize an object until all its potential supporters have been recognized. Then the potential supporter with the highest horizontal

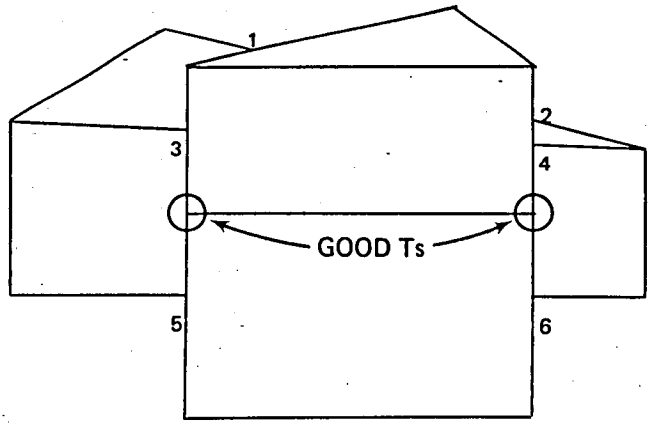


FIG. 13. Illustrating SEGMENT

surface is identified as the actual supporter for that object. The end points of all the base edges of the object can then be located in 3-space.

RECOGNIZE attempts to name an object by matching features of its line drawing against the stored properties of the prototypes. A succession of tests is applied to the prototypes until, hopefully, only one remains. If the line drawing is complete (which is determined by a simple heuristic picture topology test) then the first test looks at the number of visible faces and vertices, otherwise the topology of the complete faces is used. The second test compares lengths of base edges while the third test compares angles between the base edges. The fourth test assumes that lines vertical in the picture correspond to vertical edges if they are not labelled as base edges. The length of such an edge can be calculated and compared with the prototypes.

When the object is named and three corners of the base edges of it are located in space then the object is positioned by identifying three corresponding points on the prototype.

VERIFY predicts the picture appearance when every object has been recognized and located. If a body has more than 3 lines in the prediction that do not appear in the input or if there are any lines in the input that have not been predicted then VERIFY reports back to RECOGNIZE and asks for a new suggestion.

Falk's program is a good attempt at overcoming imperfect line data but, as he has taken from Guzman an almost total reliance on local picture-based heuristics, INTERPRET is open to the objections raised against SEE above. In fact, Falk extends their usage beyond body segmentation to include support and completion heuristics of the same general nature. To demonstrate the problems involved, we will present for each of those stages of INTERPRET a specific example of a picture where the program [at least, that version of it described in (Falk, 1972)] appears to go astray. These simple examples using only Falk's

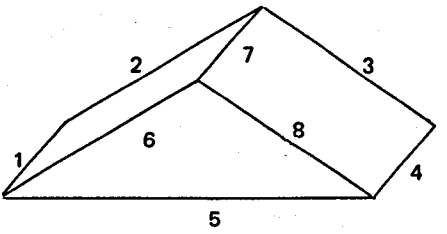


FIG. 14. Illustrating SUPPORT

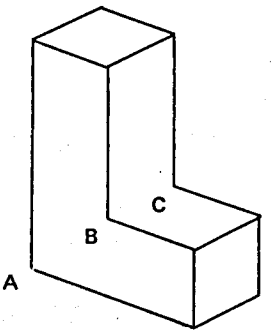


FIG. 15. Illustrating COMPLETE

prototypes are not malevolently constructed using degenerate views or unlikely alignments, nor can the problems be attributed to insufficient data as the pictures are perfect line diagrams (except for the one missing line that COMPLETE should insert).

SEGMENT finds only 2 bodies in Figure 13. It matches the back-to-back T's of the partially occluded wedge to get one body, (that is, it matches junction 1 with junction 2, 3 with 4, and 5 with 6) but the two stacked wedges in front are seen as one body because the 2 circled junctions are both classified as GOOD T.

SUPPORT eliminates line 1 of Figure 14 as a base edge of that wedge because it is a line at a downward open L vertex.

Finally, in Figure 15 there is a line missing from the picture of an L-beam. COMPLETE has a routine ADDLINE to deal with this. ADDLINE is activated by a context of a pair of L vertices with parallel sides. In Figure 15 there are two such contexts: AB and BC. The first context to be picked up is not defined but if it is AB and ADDLINE puts a line between A and B it destroys the second context, BC. Regardless of which context is found first, ADDLINE certainly has no way of knowing that line BC makes more sense than AB because in the picture domain there are no grounds for preferring one over the other; both are correct as pictures.

The remark "makes more sense" applies not to the picture itself but to what is depicted, the scene. Similar comments apply to the failures of SEGMENT and

SUPPORT and so it becomes clear that the program must have some kind of 3-dimensional interpretation before evaluating predicates such as 'same body', 'supports' and 'missing edge'. But the only way Falk has of getting a 3D interpretation is by recognizing the objects. This is a chicken and egg problem: the program needs to recognize the objects to get a 3D grip on the scene in order to recognize the objects.

The way to break this circularity is to realize that recognition, that is, the identification of an object as a particular member of a set of prototypes, is not the only way of getting a grip on the scene. There are general principles about the picture-taking process and the nature of opaque polyhedra that one can incorporate in a procedure to interpret line diagrams that does not use any specific prototypes. Huffman (Huffman, 1971) and Clowes (Clowes, 1971) working at the same time as Falk independently proposed such a procedure which can now be seen as a step towards the solution of the chicken and egg problem of scene analysis.

THE LINGUISTIC APPROACH

Before we examine that procedure, another approach to picture processing must be mentioned. In the nineteen-sixties a scattered group of people were trying to find suitable representations for picture descriptions as suggested by Minsky (Minsky, 1961). Struck by the persuasive analogy between pictures and natural language and influenced by Chomsky's (Chomsky, 1957,1965) account of syntactic structures, some, such as Kirsch (Kirsch, 1964), Ledley (Ledley, 1964), Narasimhan (Narasimhan, 1966) and Anderson (Anderson, 1968) wrote grammars for restricted classes of pictures while others such as Clowes (Clowes, 1969), Evans (Evans, 1969), Shaw (Shaw, 1969), and Stanton (Stanton, 1970) attempted more general picture description languages. Like all analogies the linguistic approach eventually collapsed and died (for the obituary notice and postmortem see (Stanton, 1972) and (Clowes, 1972a)) but it left a legacy of insights. For example, following Chomsky's emphasis on the uses of anomaly, a common technique in the linguistic approach exploited pictures of impossible objects in order to tease out the rules whereby we assign structure and meaning to pictures. Both Huffman (Huffman, 1971) and Clowes (Clowes, 1971) used this technique to examine the interpretation of line diagrams as polyhedra.

THE HUFFMAN-CLOWES LABELLING ALGORITHM

As we remarked earlier Guzman's SEE somewhat surprisingly deduces body membership of two surfaces from the appearance of the corners that they share. The most obvious question to ask is: why does it work? Another question might be: what else can we infer from the junction geometry? The answer to the latter question will indeed help us answer the former. To start with we note that it makes more sense to infer local (rather than global) scene properties from local picture evidence. In particular if we rely on the shape of junctions as evidence

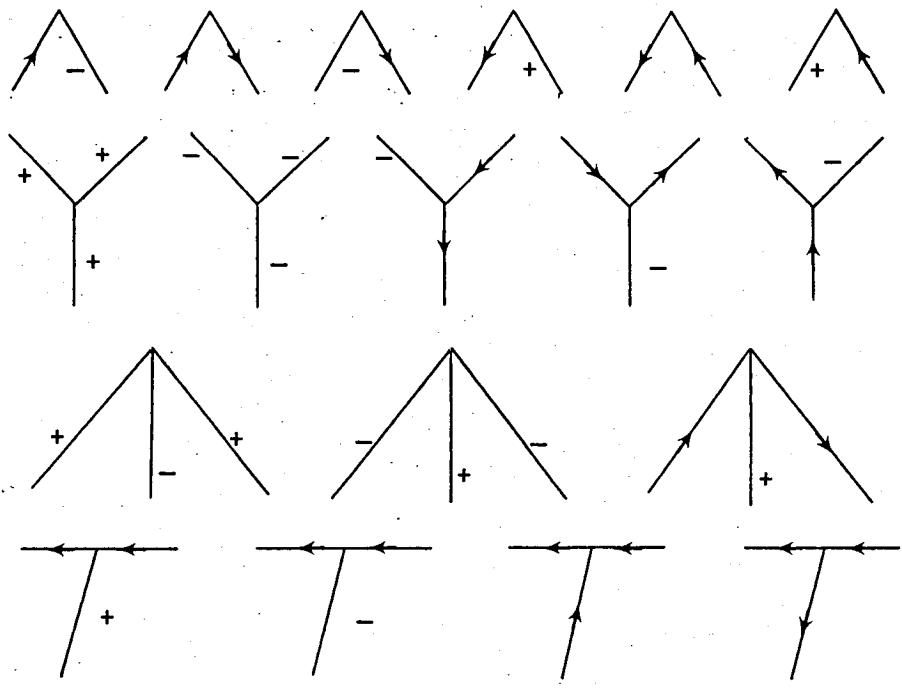


FIG. 16. The Huffman-Clowes junction interpretations

we should be making inferences about the corners they depict. Restricting themselves to 2-line and 3-line junctions and 3-surface corners, Huffman and Clowes observed that each Guzman junction category must have one of a small number of corner interpretations which are described by the predicates convex, concave and occluding which apply to the edges meeting at the corner. In Huffman's notation, + labels a convex edge with both surfaces visible; - labels a concave edge and an arrowhead labels an occluding edge that belongs to the surface on the right (as you move in the direction of the arrow). The surface on the left is behind the edge and partially occluded by the surface on the right.

Figure 16 shows the interpretations for each legal junction type (L, FORK, ARROW, and T). For all but the T these interpretations are actually corners. Considering all four possible labellings for each line gives $4^2 = 16$ for the L, $4^3 = 64$ for the others as against the reality of 6 for the L, 5 for the FORK and so on; hence, it is apparent how useful these legal corner interpretations could be. In order to use this table of interpretations the only further scene coherence rule is that an edge must have the same interpretation at both of its visible endpoints. The labelling algorithm described by Clowes starts with the background region and constructs all interpretations in parallel whereas Huffman suggested a depth-first search, backtracking when coming upon a junction that has no interpretation consistent with the labels that have already been placed on some of its lines. Both procedures not only label the edges of the scene but also recover

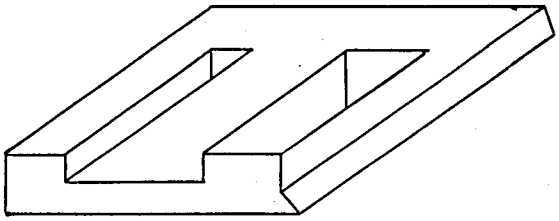


FIG. 17. An anomalous object

some of the hidden structure in that occluding edges have attached to them surfaces that are turned away from the viewing direction.

There are several reasons to judge this algorithm to be an important step foward in scene analysis. Let us start with impossible objects. There is theoretical satisfaction in having a procedure that returns no interpretations of a picture such as the one reminiscent of the devil's pitchfork, Figure 17 (taken from [Clowes, 1971]), if we ourselves cannot assign a plausible three-dimensional interpretation. But this ability would also be of practical use in a scene analysis program. Figure 9 (b), which SEE happily accepted and parsed, can be rejected as a candidate for object status because it cannot be labelled. This is a sufficient but unfortunately not necessary condition that the object be impossible as Huffman showed. But to be able to make this discrimination suggests that the method has greater descriptive power than the only other prototype-free program, SEE. A comparison of the scene description generated by this algorithm with that given by SEE shows how true that is. Here we have edges known to be convex, concave or occluding, the visible part of a surface defined by edges belonging to that surface or to another known surface and some conclusions about hidden surfaces that share an edge with a visible surface.

The question "Why does SEE work?" can now be answered in detail. Suppose that we were only concerned with convex objects, then from the set of corner interpretations used by the labelling algorithm (Fig. 16) eliminate all corners with concave edges, including those for the L that imply a hidden concave edge, leaving the set of Figure 18. Notice that the L, FORK and ARROW junctions now have unique corner interpretations. The concave edges that appear when one body abuts or rests upon another are here taken to be occluding edges as they would be if the bodies were slightly separated. In this world of convex polyhedra, convex edges (+) join surfaces of the same body while surfaces of different bodies appear at occluding edges (> and <) so using this corner set a body partitioning is easy to achieve. That's what Guzman did! The links were planted at unambiguously convex edges. The link-planting rules of Figure 8 are derived from the corner interpretations of Figure 18 by replacing + by a link and occluding by no link. The link suppression rules, "no link is placed across a line at a junction if its other end is a barb of an ARROW, a leg of an L or the crossbar of a T," can be seen from Figure 18 to suppress a link across an edge if its other end shows it to be unambiguously occluding. The accumulation of link

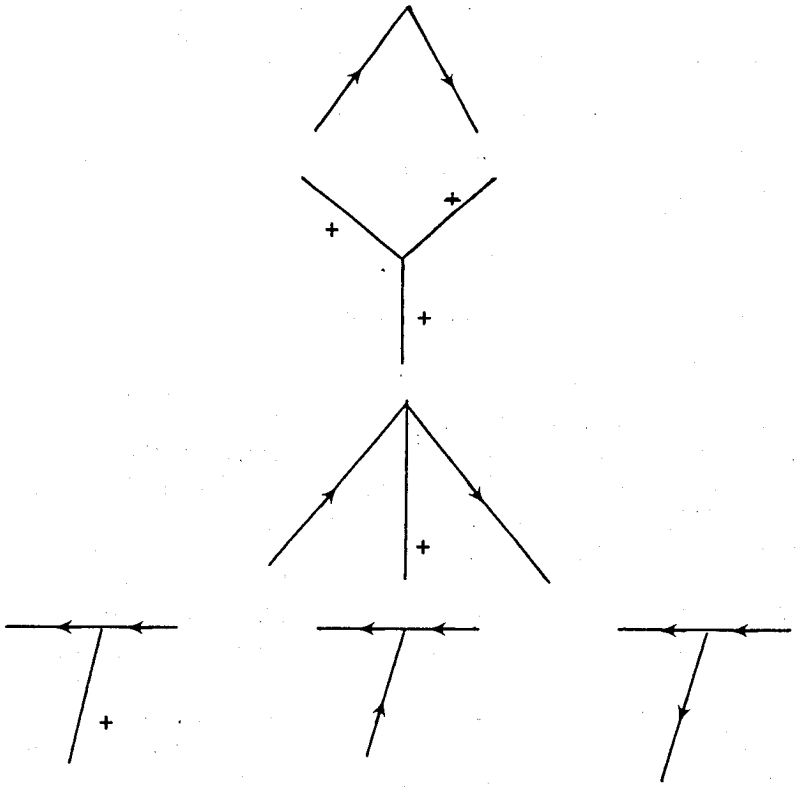


FIG. 18. The junction interpretations for convex polyhedra

evidence relies on 2 links between surfaces which means in effect that both ends of an edge must agree that it is convex for it to be so taken as in the Huffman-Clowes algorithm. If only one end says so there is a conflict which must be heuristically resolved. This provides a scene-coherent account of why Guzman's picture-based heuristics worked and incidentally explains why SEE doesn't work on concave objects (Winston, 1968).

The next step is to use the scene as labelled by the Huffman-Clowes algorithm as a more reliable basis for body segmentation. A first guess might say: the visible aspect of a body is a maximal set of surfaces joined by convex or concave edges. This isn't quite right because by that criterion the labelled cube in Figure 19 is part of the same body as the background, by virtue of the two concave edges. Such concave edges define body boundaries. Waltz (Waltz, 1972) as we shall see called them "separable" and used a further subcategorization of concave edges to solve this segmentation problem.

Returning to Falk's INTERPRET, the labelling algorithm is considerable potential help in solving the chicken and egg problem. Consider the three stages where INTERPRET was seen (above) to get into trouble: SEGMENT, SUPPORT and COMPLETE. The above discussion of a scene-based approach to body seg-

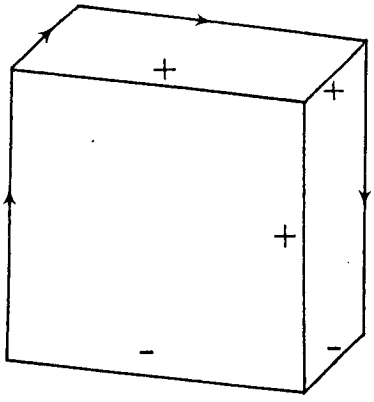


FIG. 19. An interpretation of a picture

mentation applies to the problem with SEGMENT. The specific problem illustrated in Figure 13 requires more interpretations for the T junction than shown in Figure 16 but the extension is straightforward as will be shown in the discussion of Waltz' program.

SUPPORT rejected edge 1 of Figure 14 as a potential base edge. A labelling of that picture gives edges 1, 2, 3, 4, and 5 as occluding edges and 6, 7, and 8 convex. Furthermore, edges 1, 5, and 4 are attached to a single hidden surface while edges 2 and 3 are attached to a different hidden surface of the same body. A support algorithm given that information only has to decide that the former surface is the support surface.

The first thing COMPLETE should do is decide if an edge is in fact missing. If the object cannot be labelled then that must be the case. For Figure 15 no labelling is possible as shown by the conflict at the circled junction of Figure 20 (a). That labelling for that junction is not a legal interpretation of an L (see Fig. 16). Since lines can only be added to the picture and junctions in a picture of a single body are not allowed more than three lines, a line must be added to the circled junction of Figure 20 (a) joined to either of the facing L junctions. Either of the lines AB or BC can be inserted and the picture labelled as Figure 20 (b) and Figure 20 (c) show but clearly only (c) makes sense in terms of the prototypes. This leads us to consider the matching procedures in INTERPRET. They should operate in a domain of surfaces (visible and hidden), corners and edges (convex, concave and occluding) rather than directly in the picture, as do the picture topology matching routines of RECOGNIZE and VERIFY. Besides being more sensible, matching in the scene domain is also clearly more efficient because the program has richer structures to compare. For example, a match could be quickly aborted in the scene domain if an edge were of the wrong type.

The labelling algorithm does not sweep away all the difficulties in Falk's program but it points in the right direction; however, there are some problems with the labelling algorithm as described here. It can make mistakes. In Figure

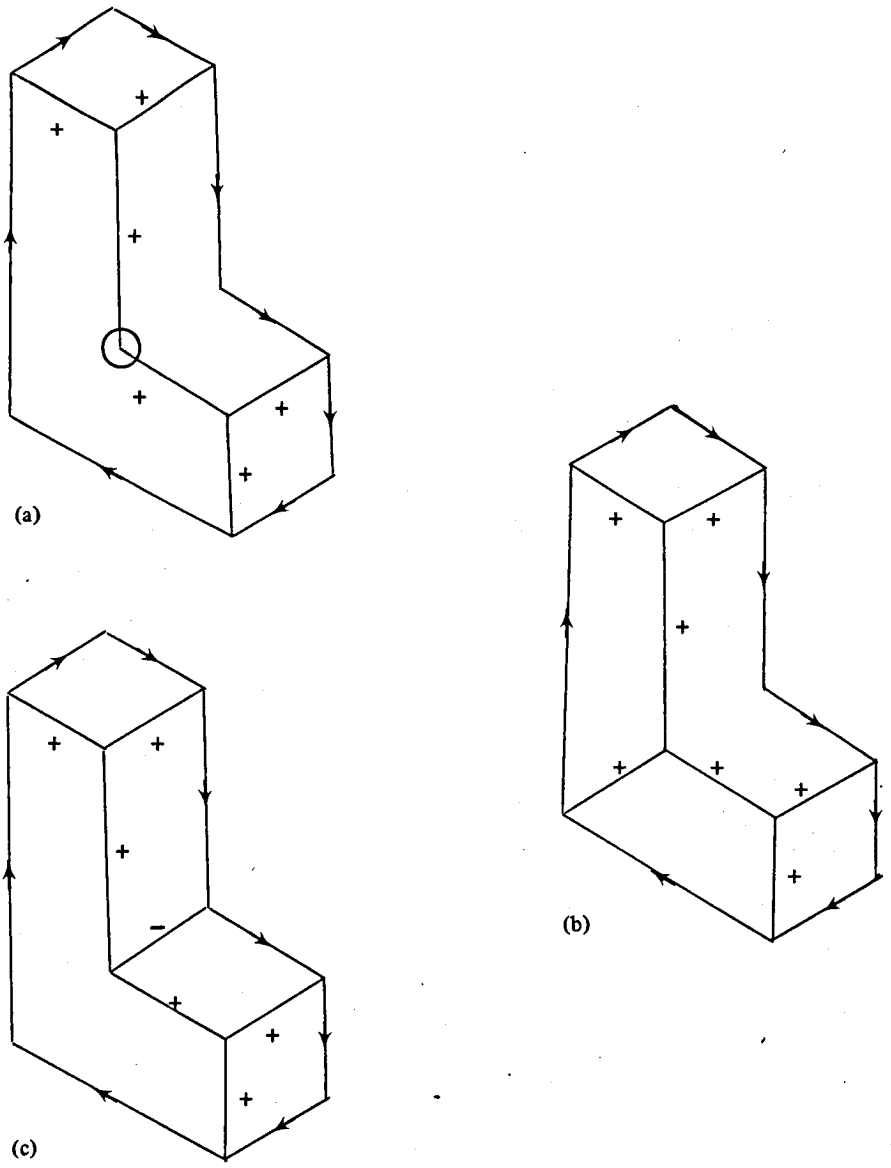


FIG. 20. Completing a picture of an object

21 (a) it incorrectly labels a legitimate view of a cube (it will of course produce all the correct labellings as well) and in Figure 21 (b) (adapted from (Huffman, 1971)) it labels an object that cannot be a polyhedron with planar surfaces. Both sorts of mistakes can be avoided by an extension of the labelling algorithm: if two lines (a and b) shared by a pair of regions (A and B) are not collinear then the lines cannot both depict convex or concave edges. But that *ad hoc* extension

evades the key issue which is that the algorithm has no requirement that surfaces be planar nor is there any way that it can be systematically introduced without radical changes in the algorithm. Beyond saying that a surface cannot change from visible to hidden (unless, of course, it is partially occluded), there is no coherence required of a surface. This can be further illustrated by noting, as Huffman did, that the algorithm finds a labelling for the impossible triangle of (Penrose and Penrose, 1958). That object can only be realized if some of the surfaces are highly skewed.

In order to handle some other problems which arise such as many-surface corners, alignments of bodies in the scene, coincidence of viewing direction and object surfaces, shadow edges and so on, does one simply add *ad semi-infinitum* to the lists of corner interpretations? Waltz has shown that that is in fact a partial answer to those problems.

WALTZ' EXTENSION OF THE LABELLING ALGORITHM

Waltz made two important contributions to the labelling algorithm. He expanded the set of line labels from the four used by Huffman-Clowes and he improved the mechanism of search for coherent interpretations.

His first addition to the set of possible edges was the crack—a flat edge. Next, he noticed that the visible boundaries of objects usually appear at occluding or concave edges or at cracks. To account for this he subdivided the concave and crack edge categories into separable and non-separable. An edge is separable if two or three bodies meet there. All cracks are separable but some concave edges are internal edges of a body. A separable edge has, in addition to its concave/crack label, labels that show the status of the edges of the separate bodies.

The other expansion of edge possibilities derives from a crude account of lighting. Assuming a single concentrated light source then surfaces are either illuminated, turned away from the light (self-shaded) or shaded by a shadow cast by another surface. Waltz expanded the line labels to give the illumination status of the two surfaces appearing at the edge and allowed lines to depict shadow boundaries as well as real edges. The number of possible line labels has increased from the original 4 to 53.

Following a graphical representation used by Winograd (Winograd, 1972) to depict the networks of features associated with grammatical units by his systemic grammar, we can more easily see the structure of the set of possible interpretations of a line in the network of Figure 22. In that network the choice of illumination status for each surface has not been shown so there are only 11 distinct line interpretations.

Turning to the possible corners and their picture appearance, Waltz used the Huffman-Clowes junction categories and also all 4-line and some 5-line junctions. Following a straightforward procedure, Waltz considered all possible object configurations viewed and lit from all possible octants to generate the possible corners list for each junction category. The length of the corner list for each category varies from 10 to 826 with a grand total of 3256. The actual corners

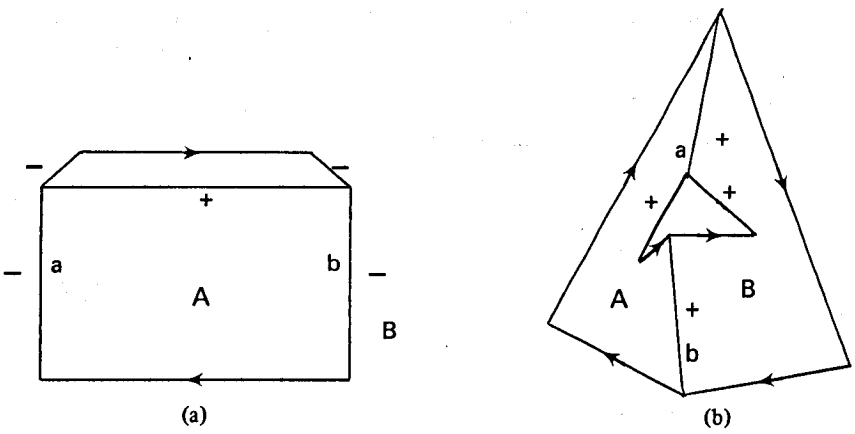


FIG. 21. Labelling problems: (a) an anomalous interpretation of an object; (b) an interpretation of an anomalous object

are all either trihedral or formed by more than one convex trihedral object but he also includes some interpretations of junctions formed by accidental alignments in the scene.

With so many possible corners for each junction, Waltz realized that time and space limitations rule out a simple depth or breadth-first search, so he devised a more efficient two pass procedure. The first pass through the junctions, the filtering procedure, is a modified breadth-first search that weeds out the possible corner list for each junction by checking in the lists of every adjacent junction that has previously been processed for at least one corner with the same label for the connecting line. If that check is not successful then that possible corner is weeded out of the list for that junction. This discarding causes the program to reconsider junctions it has already looked at so the discarding action may have an effect that propagates through many junctions. Since this procedure does not actually construct complete interpretations as it goes, it need not find *all* pairs of corners with the same label for the connecting line as Clowes' procedure does; hence, it avoids 'the intermediate expression bulge' of the earlier procedure. This weeding process drastically reduces the possible corner lists so that the second pass can easily backtrack to find complete interpretations without requiring exponential time as Huffman's procedure does. For extensions and generalizations of this and related algorithms see (Mackworth, 1975).

Figure 23 shows a typical scene labelled by Waltz' program. The convex and occluding edges are shown as they were for the Huffman-Clowes labelling. The concave edges here are separable so they are additionally labelled with an occluding arrowhead indicating the sense of occlusion the edge would have if the object were picked up. Cracks are labelled with a C and a similar occlusion arrowhead. Shadow boundaries are shown with arrows pointing across the line into the shadowed region.

Waltz' achievement was to show that the labelling technique can be extended

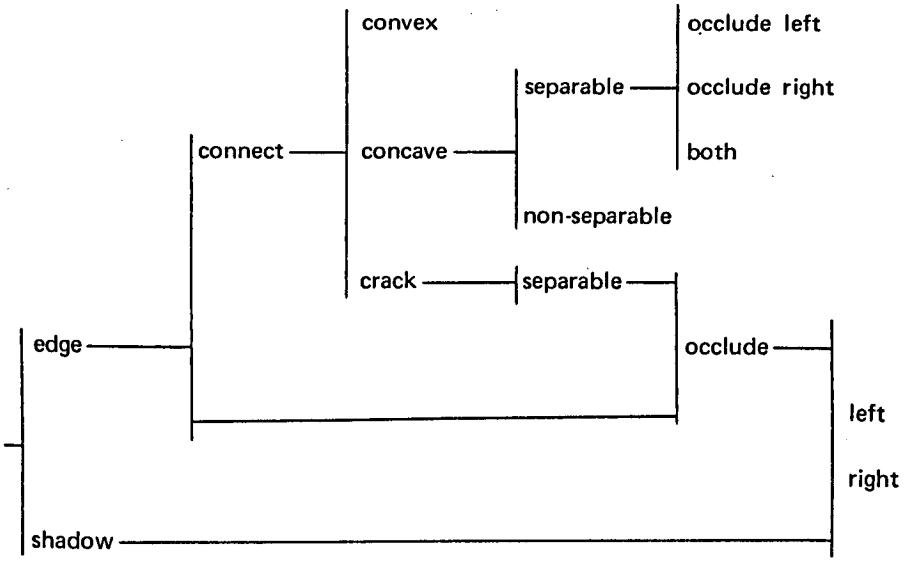


FIG. 22. A network of the interpretations of a line

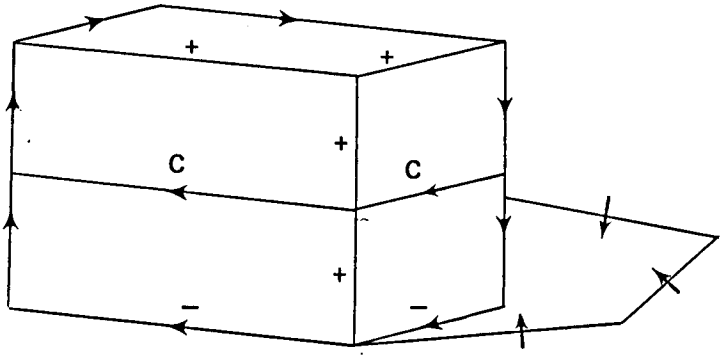


FIG. 23. A scene labelled by Waltz' program

to handle more realistic scenes than previously although it has yet to be incorporated in a scene analysis program using grey scale picture data. Most of the remarks made above about the Huffman-Clowes procedure apply equally to Waltz' extension of it. In particular, the twin problems of anomalous interpretations of legitimate scenes and acceptance of impossible objects demonstrated in Figure 21 for the earlier procedure still remain. In fact, there is a further scene (Fig. 24) to which Waltz' program assigns the anomalous interpretation shown. But this anomaly cannot be avoided by the simple strategem suggested to cope with the problems of Figure 21 because the requirement that the common edges of intersecting surfaces appear collinear is satisfied here. What is required to reject this anomaly is a chain of reasoning involving hypotheses and deductions about surface and edge orientations. It is left to the reader to construct the argument.

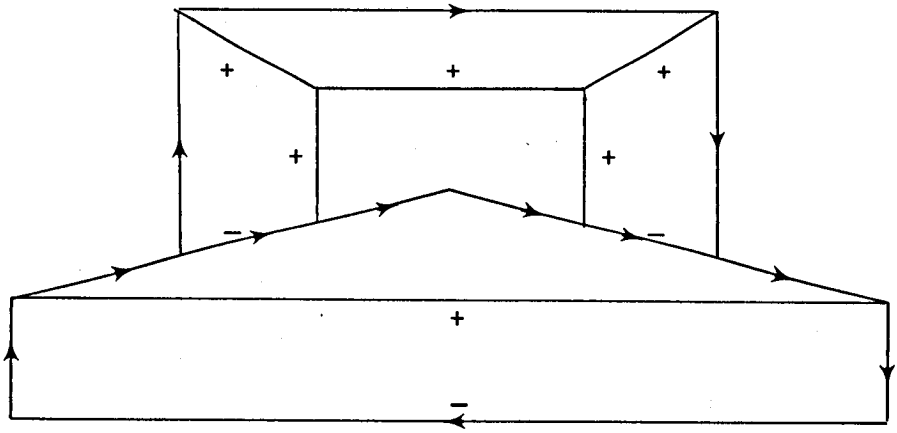


FIG. 24. An anomalous interpretation of a scene

The form of Waltz' input assumes the ability to see every edge perfectly including all those inside the shadow regions even though there is only a single light source (Fig. 23). Is this having your shadow cake and eating it too? Waltz does consider simple cases of missing edges, but, as he emphasized, the labelling technique uses only the topology of the line drawing and local junction shape information. He gives many good examples of pictures equivalent on that basis that seem to require very different interpretations or missing edge completions.

As we pointed out in the criticism of the Huffman-Clowes algorithm an interpretation procedure for line drawings must use more than the picture topology and agreement between adjacent corners if it is to be satisfactory in its treatment of all the various aspects of scene analysis discussed above.

POLY: EXPLOITING SURFACE COHERENCE AND THE EDGE HIERARCHY

One approach that can only be briefly mentioned here is the author's program POLY (Mackworth, 1973,1974a). Using a representation for surface orientations suggested by Huffman (Huffman, 1971), the gradient space, POLY hypothesizes and makes inferences about surface and edge orientations and positions exploiting heavily the hierarchical structure of the network of interpretations of a line (see Fig. 22; the version of POLY implemented did not make the shadow or separable edge distinctions) thereby dispensing with the lists of possible corners. The only backtracking search in POLY is at the connect/occlude level of distinction in the edge hierarchy; the other features of the edges are then inferred directly from the surface, edge and corner representations used. While the size of the underlying search space has been drastically reduced, the resulting interpretation is richer in descriptive power including as it does relative information on surface and edge orientation and position. This descriptive adequacy or higher level of scene coherence not only makes the interpretation more useful

but also ensures that the anomalies of Figure 21(a), Figure 21 (b) and Figure 24 do not arise.

CONCLUSION

In a paper on descriptive languages and problem solving Minsky (Minsky, 1968) sees artificial intelligence as an attempt to achieve adequate descriptions and procedures for manipulating them for specific task domains. This view provides the best framework for understanding the first decade of scene analysis. Starting with Roberts, there has been a continual struggle to achieve adequate picture and scene descriptions and procedures for relating the two with considerable progress being made. But, *pace* Chomsky, descriptive adequacy is not enough. The representation issue may be in a reasonably satisfactory state but the control issue is not. Of the work described here, only Roberts and Waltz have paid it sufficient attention. Of work not described here for space reasons, MIT's COPY DEMO (Winston, 1973) and, more recently, Shirai's context-sensitive linefinder (Shirai, 1973) are the most adequate from that viewpoint. Shirai's program, for example, uses a procedural model of the picture that is essentially a very loose characterization of all line drawings of scenes of convex polyhedra to direct the image analysis which consists of line and junction detection in grey-scale pictures. If we dare risk a linguistic analogy, that appears to be a syntactic model while we have an entire spectrum of semantic models ranging from Falk's size-specific polyhedral prototypes through Robert's transformable prototypes, the architectural models of Winston's thesis (Winston, 1970), the Guzman-Huffman-Clowes-Waltz corner models, the hierarchy of line interpretations, to size or shape-specific surface models (Mackworth, 1974b).

If we choose the active model of perception suggested to us by Roberts' program, how are we to cope with this abundance of models? How do they sensibly interrelate? How should they be invoked? When should they be invoked? And yet cope we must, for surely the availability of a wide variety of effective schemata conjoined with the ability to invoke the relevant subset of them at the appropriate time is the hallmark of intelligence.

ACKNOWLEDGMENTS

The author is indebted to Max Clowes for inspiration and sound criticism. Robin Stanton, Stuart Sutherland, and Aaron Sloman also helped shape these views on vision. This work was supported by the National Research Council of Canada and the Science Research Council of Great Britain.

REFERENCES

Anderson, R.H. (1968) Syntax-directed recognition of hand-printed two-dimensional mathematics. *Ph.D. Thesis,* Division of Engineering and Applied Physics, Harvard University.

Bartlett, F.C. (1967) *Remembering,* Cambridge University Press.

Chomsky, N. (1957) *Syntactic Structures,* Mouton and Co., The Hague.

Chomsky, N. (1965) *Aspects of the Theory of Syntax,* M.I.T. Press, Cambridge, Mass.

Clowes, M.B. (1969) Transformational grammar and the organization of pictures. *Automatic Interpretation and Classification of Images,* (ed. Grasseli, A.), Academic Press, N.Y., pp. 43-77.

Clowes, M.B. (1971) On seeing things. *Artificial Intelligence,* 2, 1, 79-112.

Clowes, M.B. (1972) Scene analysis and picture grammars. *Graphic Languages,* (eds. Nake, F. and Rosenfeld, A.), North-Holland, Amsterdam, pp. 70-82.

Evans, T.G. (1969) Descriptive pattern analysis techniques. *Automatic Interpretation and Classification of Images,* (ed. Grasselli, A.), Academic Press, N.Y., pp. 79-96.

Falk, G. (1972) Interpretation of imperfect line data as a three-dimensional scene. *Artificial Intelligence,* 3, 2, 101-144.

Gregory, R.L. (1974) *Concepts and Mechanisms of Perception,* C. Scribner's and Sons, N.Y.

Guzman, A. (1968) Decomposition of a visual scene into three-dimensional bodies. *AFIPS Proc. Fall Joint Comp. Conf.,* 33, pp. 291-304.

Huffman, D.A. (1971) Impossible objects as nonsense sentences. *Machine Intelligence* 6, (eds. Meltzer, B. and Michie, D.), Ediburgh University Press, Edinburgh, pp. 295-323.

Kirsch, R.A. (1964) Computer interpretation of English text and picture patterns. *IEEE Trans. on Electronic Computers,* EC13, 363-376.

Ledley, R.S. (1964) High speed automatic analysis of biomedical pictures. *Science,* 146, 216-223.

Mackworth, A.K. (1973) Interpreting pictures of polyhedral scenes, *Artificial Intelligence,* 4 2, 121-137.

Mackworth, A.K. (1974a) On the interpretation of drawings as three-dimensional scenes. *D. Phil Thesis,* Lab. of Exp. Psych., Univ. of Sussex.

Mackworth, A.K. (1974b) Using models to see. *Proc. AISB Summer Conf,* pp. 127-137.

Mackworth, A.K. (1975) Consistency in networks of relations. Tech. Rep. 75-3, Dept. of Computer Science, Univ. of British Columbia.

Michie, D. (1974) On not seeing things, in *On Machine Intelligence,* Wiley.

Minsky, M.L. (1961) Steps toward artificial intelligence. *Proc. Inst. Radio Engineers,* 49, pp. 8-30.

Minsky, M.L. (1968) Descriptive languages and problem-solving. *Semantic Information Processing,* (ed. Minsky, M.L.), M.I.T. Press, Cambridge, Mass., pp. 419-424.

Minsky, M.L. and Papert. S. (1972) Progress Report, *A.I. Memo 252,* M.I.T., Cambridge, Mass.

Minsky, M.L. (1975) A framework for representing knowledge, in (Winston, 1975).

Narasimhan, R. (1966) Syntax-directed interpretation of a class of pictures. *CACM* 9, 3, 163-173.

O'Gorman, F. and Clowes, M.B. (1973) Finding picture edges through collinearity of feature points. *Proc. Third Int. Joint Conf. on Artificial Intelligence,* Stanford Research Inst., Stanford, Calif., pp. 543-555.

Penrose, L.S. and Penrose, R. (1958) Impossible objects: a special type of illusion. *Brit. J. Psych.,* 49, 31-33.

Roberts, L.G. (1966) Machine perception of three-dimensional objects. *Optical and Electro-optical Information Processing,* (eds. Tippet *et al.*), M.I.T. Press, Cambridge, Mass., pp. 159-197.

Shaw, A.C. (1969) A formal picture description scheme as a basis for picture processing systems. *Information and Control,* 14, 1, 9-52.

Shirai, Y. (1973) A context-sensitive line finder for recognition of polyhedra. *Artificial Intelligence,* 4, 2, 95-119, also in (Winston, 1975).

Southall, J.P.C. (1962) (ed.) *Helmholtz' Treatise on Physiological Optics,* Vol. III, Dover.

Stanton, R.B. (1970) Computer graphics–the recovery of descriptions in graphical communication. *Ph.D. Thesis,* Dept. of Electronic Computation, University of New South Wales.

Stanton, R.B. (1972) The interpretation of graphics and graphic languages. *Graphic Languages,* (eds. Nake, F. and Rosefeld, A.), North-Holland, Amsterdam, pp. 144-159.

Waltz, D.L. (1972) Generating semantic descriptions from drawings of scenes with shadows. *MAC AI-TR-271*, M.I.T., Cambridge, Mass., also in (Winston, 1975).

Winograd, T. (1971) Procedures as representation for data in a computer program for understanding natural language. *MAC AI-TR-84,* M.I.T., Cambridge, Mass.

Winston, P.H. (1968) Holes, *A.I. Memo 163,* M.I.T., Cambridge, Mass.

Winston, P.H. (1970) Learning structural descriptions from examples. *MAC AI-TR-76,* M.I.T., Cambridge, Mass., also in (Winston, 1975).

Winston, P.H. (1973) The MIT robot. *Machine Intelligence* 7, (eds. Meltzer, B. and Michie, D.), Ediburgh University Press, Edinburgh, pp. 431-463.

Winston, P.H. (1975) *The Psychology of Computer Vision*, McGraw-Hill.

WORLD-KNOWLEDGE FOR LANGUAGE-UNDERSTANDING

26

Inference and Knowledge in Language Comprehension

Eugene Charniak

Institute for Semantic and Cognitive Studies
Castagnola, Switzerland

INTRODUCTION

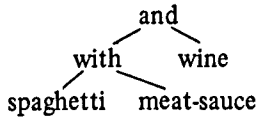
Why make inferences?

To use language one must be able to make inferences about the information which language conveys. This is apparent in many ways. For one thing, many of the processes which we typically consider "linguistic" require inference making. For example, structural disambiguation:

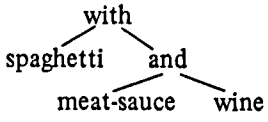
(1) Waiter, I would like spaghetti with meat sauce and wine.

You would not expect to be served a bowl of spaghetti floating in meat sauce and wine. That is, you would expect the meal represented by structure (2) rather than that represented by (3).

(2)



(3)



Note however that if you had asked for spaghetti with butter and garlic you would have wanted structure (3). So to get your order right the waiter must have made an inference based on his knowledge of food.

The same is also true for word sense disambiguation and reference determination, as seen in examples (4) and (5) respectively.

(4) The box was in the *pen*. [not the writing implement]

(5) At the supermarket Fred found the shelf where the milk was. He payed for *it* at the checkout counter and left. [Not the shelf but the milk. And to be precise, not the "milk" mentioned in the story, which is in some sense all the milk on the shelf, but a single carton of milk.]

Furthermore, any task which requires that a person understand language invariably requires the making of inferences. Some obvious examples are: question answering (What did Fred buy?); translation (Translating (5) into German requires finding the referent for "it" since "shelf" and "milk" have different genders); summarizing (It's hard to give a short example of why inference is needed here, but the general idea behind summarizing a story is to find the "main thread". However, the connecting links of the thread are generally not stated explicitly, but rather have to be inferred).

The need for knowledge

In each of the above examples, in order to make the necessary inference we needed information beyond that provided by the example. So in (5) we need to know that shelves are usually not purchased at supermarkets, and that one would not generally buy all the milk which the supermarket had on its shelves. In example (1) we must know about wine and its use as a beverage, and that while it is sometimes used as an ingredient for food, one typically does not specify the ingredients of a dish to a waiter, but rather chooses something from the menu. This paper, then,is about how *knowledge* is used to make *inferences* in the comprehension of language.

Five questions

A complete solution to this problem would require answering the following five questions. I will only state them briefly here. Their precise meaning will be best understood by the repeated reference to them throughout the paper.

1. SEMANTIC REPRESENTATION. What concepts, and in what combinations are needed to record our impressions of the world? This subject will only be referred to peripherally here.
2. INFERENCE TRIGGERING. Under what circumstances, and for what reasons do we make inferences? For example, when do we make an inference, when new information comes in, or when a question is asked? We also make inferences to help determine reference. Does this affect our answers to any of the other questions?
3. ORGANIZATION. Given we want to make an inference, how do we locate the needed information? The all purpose response here is that we have "pointers" to the information. The question then becomes how do we organize these pointers? So we might, for example, imagine that certain pieces of information, like that about supermarkets, are "grouped" together. What "grouping" might mean is that there is a pointer from some topic to each piece of information.
4. INFERENCE MECHANISM. Once we have located a fact, how do we know how to use it? For example, if facts were represented as programs, the interpreter for the program would tell us how to use the fact.
5. CONTENT. What is the knowledge which we have of the world that

enables us to understand language? Note that while the answers to the other questions would hopefully be culture independent (with the possible exception of (1)), the knowledge we have is clearly quite culture dependent, and even idiosyncratic.

FIRST ORDER PREDICATE CALCULUS

One set of answers, or at least partial answers, is given by assuming that the first order predicate calculus (FOPC) can be used as a theory of inference and knowledge. I will assume the reader is familiar with FOPC (and more specifically resolution theorem proving) and go on to consider what answers it gives to these questions.

1. SEMANTIC REPRESENTATION. Does FOPC tell us what predicates are needed to represent our impressions of the world? Well, for the most part the answer must be "no". We have seen that FOPC leaves us free to use whatever predicates we choose, with only a few restrictions. So it does not say if we should have a predicate TRADE(w,x,y,z) which means that w gave x to y for z, or if we should have a more basic, or "primitive" predicate. For example, Schank (Schank, 1973) argues that there is a more primitive verb, which he calls "ATRANS" which underlines not only "trade", but also "buy", "sell", "give", "take", etc. In particular, he would roughly represent TRADE(w,x,y,z) as

(6)	ATRANS (w,w,x,y)	[(w was the actor in transferring x from w to y)]
(7)	ATRANS (y,y,z,w)	[(y gave z to w)]
(8)	CAUSE ((6),(7))	
(9)	CAUSE ((7),(6))	[(6) and (7) mutually caused each other)]

(Technically statements (8) and (9) as written are not legal FOPC statements, but there are well known ways to change things so that they are legal.) So the only answer FOPC gives to the question is that certain items like AND, OR and NOT are useful to use, but otherwise you can do much as you choose.

2. INFERENCE TRIGGERING. What does FOPC say about when and why we make inferences? Well, it is hard to be too firm about this, since one can always change things slightly, but the general idea behind FOPC is that one only makes inferences when one is asked a question. At the moment, this may not seem like much of a limitation but we will see that it is.
3. ORGANIZATION. How do we locate a needed fact in FOPC? FOPC in general, and resolution theorem proving in particular, say very little on the subject. The result has been the inability of systems

based on FOPC to handle large data bases, or complex inferences because the system gets lost in the combinatorial explosion.

4. INFERENCE MECHANISM. Given that we have located the facts we intend to use, does FOPC tell us what to do with them? Here the answer is an unequivocal "yes", especially when we are talking about resolution theorem proving. In resolution theorem proving, to produce a new fact from old there is only one thing you can do with the old facts–resolve them together. When you come right down to it, FOPC is primarily a theory of inference mechanism.
5. CONTENT. FOPC says nothing about what facts one needs to know about the world. It turns out that there are some facts which are difficult, if not impossible, to express in FOPC, but aside from that FOPC makes no suggestions.

To summarize, FOPC says a lot about inference mechanism, and a little about semantic representation. As we see then, one problem with FOPC is that it does not provide answers to most of our questions.

WHEN DO WE MAKE INFERENCES

And there are still other problem with FOPC. To understand them however it will be necessary to take a close look at the circumstances under which we will want our inference systems to make inferences. In particular we want to answer the simple question, when do we make an inference?

Let us assume we are talking about a question-answering system which accepts information expressed in a natural language and will answer questions based on it. Then there are two obvious times when we might make an inference. The first is when a question is asked which requires the inference be made. The second is that point in the input when the system has been given enough information to make the inference. We will call these two possibilities "question time" and "read time", respectively.

There are obvious advantages to making inferences only at question time. Given a particular set of facts there are an incredible number of inferences which might be drawn (remember the combinatorial explosion). We clearly cannot make all possible inferences, so by waiting until a question is asked we guarantee that we will only be making those inferences which we must make in order to answer the system user's question. However, there are good reasons why this is a bad thing to do.

As a psychological question

If we wanted to build a model which simulated people reading a text, we could ask the purely empirical question, do people make inferences about the text while they read. In a typical psychological experiment on the question, a subject will be given a piece of text to recall, only to have him "remember" facts which were not present in the text, but which were inferred from the text. To me this strongly suggests that the inferences were made at read time, although

the psychologists note that the inferences could have been made at recall time.

Is it even possible to postpone all inference?

But even if we were not interested in a psychological model, we have already seen reasons why our model could not postpone all inference making until question time. It is not possible to do word sense or structural disambiguation, or noun phrase reference without making inferences. We saw several examples of this at the start of the paper, let me give one more here.

(10) Today was Jack's birthday. Janet and Penny went to the store. They had to get presents. "I will get a top," said Janet. "Don't do that," said Penny. "Jack has a top. He will make you take it back."

The problem here is that the "it" in the last sentence does not refer to the last mentioned inanimate object (Jack's top) but rather the second to last (the top Janet was thinking of getting). To get this reference correct one must make use of one's knowledge about exchanging things at stores.

Problem occasioned inference

Considerations like this have produced a consensus that some inference must be done at read time. The question now becomes how much and of what sort. A useful distinction in this regard is that of Wilks (Wilks, 1975) between "problem-occasioned" and "non-problem-occasioned" inference. A problem-occasioned inference is one which we perform in order to accomplish the translation into internal representation. It is called "problem-occasioned" because we perform such inference only when we run into a problem like an ambiguous word, or a noun phrase whose referent is in doubt. A typical example of a non-problem-occasioned inference is exhibited by the story "Janet shook her piggy-bank. There was no sound". The inference that there is nothing in the piggy-bank is non-problem-occasioned since no ambiguity in the story required this inference for its resolution.

It is my opinion that non-problem-occasioned inferences are needed even to perform the translation into internal representation. To see why I hold this view, consider the following text about chimpanzees taken from the book *In the Shadow of Man* by Jan van Lawick-Goodall.

(11) When Flint was very small his two elder brothers, although they sometimes stared at him, paid him little attention. Occasionally while *he* was grooming with his mother, Faben very gently patted the infant.

The problem here is to figure out who the underlined "he" refers to. In fact if you have not read the book it is doubtful that you could do so, but those who come across passage (11) in the course of reading the book have no trouble because they know fact (12):

(12) Baby chimpanzees do not groom.

This fact is of course covered earlier in the book, and using it we conclude that "he" cannot refer to Flint, who is a baby, but rather to Faben.

But suppose we were only doing problem-occasioned inferences. While I do not remember how fact (12) was introduced, it is quite likely that there was no statement in the book which directly stated (12). This could happen, for example, if the text said "Young chimpanzees do not groom with non-relatives. We later learned that the young chimpanzees' behavior is the same with relatives". So while on one hand the system would have to infer (12) by making inferences on earlier parts of the text, the problem is even worse because nothing in (11) tells us that grooming is the clue. It is true that we are told that "he" was grooming with his mother, but we are also told that the grooming was with "his mother", and that Flint has two elder brothers, that Faben patted Flint, etc. The combination of both of these problems makes it hard for me to imagine how a system which only performed problem-occasioned inference could handle (11). Furthermore, I think such examples are more common than we might think. In particular, I am prepared to argue that (10) (the "take it back" story) exhibits exactly the same problem although the argument showing that this is the case is quite complex.

Question answering and problem-occasioned inference

Finally, I suspect that it will also be extraordinarily difficult to do question answering on complex stories unless the system performs non-problem-occasioned inference. Consider the following story.

> (13) Janet wanted to trade her coloured pencils for Jack's paints. Jack was painting a picture of an aeroplane. Janet said to him, "Those paints make your aeroplane look bad."

Notice that a system which only makes problem-occasioned inference would not note at read time that Janet's comment is less than unbiased truth. But suppose we asked such a system whether Janet believed what she said in this instance. How would the system decide to say no? Presumably the system would have a rule to the effect that we assume a person believes what he says unless we have reason to believe otherwise. But how would we show that Janet had reason to believe otherwise? The number of possible reasons why a person might lie seems too large to try to look for each one in the story. It would seem much more to the point to start with the facts in the story and infer that Janet has reason to lie (rather than vice-versa). But while the crucial lines in (13) occurred just prior to Janet's statement, in a real story the separation could be much larger, so it is again hard to imagine how the system could know where to look for the needed information if it had not been performing non-problem-occasioned inferences.

Implications

Assuming then that our program will be making inferences as it reads, what does this say about the system we use to make inferences?

1. CANNOT HAVE ONLY QUESTION DRIVEN INFERENCE. As we noted earlier, FOPC is a question driven system for making inferences. But we saw that not all inferences will be due to user questions, and even if we broaden our concept of question to include what we have called problem-occasioned inference, it is likely that many of our inferences will still not be question driven. On the other hand, we could not make all possible inferences at read time, there are simply too many. Hence our system must be able to distinguish between those inferences which are "important" and hence should be made immediately, and those which are not, and hence could be postponed until (and if) the need arises. For example, in story (10) it would seem reasonable to postpone the inference that Janet has lungs and a heart.

   This is not to say, however, that these problems could not be overcome in a FOPC system. Perhaps it will be possible to come up with some set of all purpose questions like "Why was I told that?" and "Why did that happen" such that a FOPC system could handle "data driven inference" by stopping after every line and asking each of the questions. But one is entitled to be suspicious of a system when it is necessary to circumvent its natural properties. Besides, in stopping to ask questions like this one has already moved some distance from a pure FOPC system.

2. TOLERANCE FOR CONTRADICTION. It is not hard to see that any system which is making inferences as the story comes in is bound to make a mistake now and then. For example, instead of story (10) where we assumed that Janet and Penny were going to the store in order to buy presents we could have had:

   (14) Today was Jack's birthday. Janet and Penny went to the store. They had to get presents. They had decided to make the presents and needed glue from the store.

   That is to say, while it is reasonable to assume that they are going to buy presents at the store, it is not an absolute certainty. So in (14) the new information must be seen as contradicting our inference, so that the inference must be withdrawn. So our system must be able to cope with seeming contradictions. However, this is another weakness of FOPC. In fact, it is a well known property of FOPC that anything can be proved from a contradiction. To see this in the resolution theorem prover we outlined, all we have to do is note that if both A and NOT A are clauses, all we have to do is resolve them and we get the null clause, hence proving whatever we set out to prove. Again, one might come up with various ways to get around this problem, but it is once again the case that the inherent properties of FOPC do not seem well suited to the task at hand.

PLANNER

What then are the alternatives to FOPC? One possibility is just to use the natural properties of LISP or some other programming language to make inferences. This is, in fact, what was done in some of the early question answering programs, for example Raphael's SIR (Semantic Information Retrieval) (Raphael, 1968).

SIR

To explain what Raphael did, it is necessary to back up a minute and explain LISP property lists. In LISP every atom has something called a property list. So, if we wanted to say that a nose is part of a person, one could represent this in LISP in the following way.

PERSON —property list→ (SUBPART(NOSE))
NOSE ——————→ (SUPERPART(PERSON))

That is, the property list is just a list of pairs, where the first item is the property, and the second is the value of the property. If we wanted to add that a heart was also part of a person, and that girls were a subclass of people we would add on:

HEART ——————→ (SUPERPART(PERSON))
GIRL ——————→ (SUPERSET(PERSON))

and we would change the property list of PERSON.

PERSON ——————→ (SUBSET(GIRL) SUBPART(HEART NOSE))

Raphael did exactly this. He had programs which translated a few kinds of sentences into such property list structures, and then he wrote programs which searched these structures in order to answer questions. So if we asked "Does a girl have a heart," the program for SUBPART would first look to see if there was a SUBPART property on GIRL, and failing this would note that GIRL was a subset of PERSON so it would look to see if PERSON had a SUBPART property. This of course would succeed in the above case and the program would respond YES.

But LISP has problems as an inference system. Raphael at the end of (Raphael, 1968) notes that SIR was becoming unmanageable, because when he wanted to add some new facility, it often required rewriting some of the old property list search routines. He then suggests that FOPC would be the solution to this problem. Indeed it would, but as we have already seen, at the cost of giving us many new problems.

An alternative is to make programming languages more suited to the needs of inference making. This is exactly what Hewitt did when he designed the programming language PLANNER (Hewitt, 1969). How PLANNER has much in

common with unicorns: we know quite a bit about it, but it never existed. That is, no language called PLANNER has yet been implemented. Nevertheless, the idea of PLANNER has been quite influential, probably due to the fact that Winograd used a hastily implemented subset of PLANNER called MICRO-PLANNER (Sussman, *et al.*, 1971) in his now famous SHRDLU program. As with FOPC, I will assume the reader is familiar with at least the rudiments of PLANNER.

PLANNER vs. LISP

What exactly does PLANNER offer over LISP as a language for making inferences?

1. DATA BASE MANAGEMENT. As we saw in SIR, it is possible to construct a data base in LISP, but the functions ASSERT, GOAL, ERASE make it much easier to do it.
2. PATTERN MATCHING MICRO-PLANNER offers a primitive pattern matching facility, so we can pull things out of the data base by knowing part of the pattern. We can bind variables at the same time. PLANNER, if it is ever written, will have a much more sophisticated pattern matching facility.
3. PATTERN DIRECTED INVOCATION. Theorems could be called on the bases of their pattern. In effect this allows us to write a function which calls a second function without ever knowing the name of the second function, but rather what it is supposed to do (insofar as we can represent this in the pattern of the theorem).
4. BACK TRACKING. This is one of the controversial features of PLANNER, but there seems little doubt that used in moderation backtracking is a useful feature. For example, one use made of it in SHRDLU occurs when the reference procedure wishes to locate a "big blue block". The general idea is to set up three goals, the first looking for a block, the second checking that it is blue, and the third checking for size. It makes sense in the case like this to use simple back tracking because there is no way to "guess" in advance which block B1, B2 or B3 say, is the one which will have all three properties. The criticism of back tracking is that it tends to encourage the construction of programs which depend too heavily on blind search. This criticism is well taken, and it was clearly a design mistake in MICRO-PLANNER that it is almost impossible to turn back tracking off. But there are cases where the search must be blind, and back-tracking is a good thing to have in such cases.

PLANNER vs. FOPC

PLANNER does not suffer from the problems of FOPC. In particular:

1. COMBINATORIAL EXPLOSION. It is not hard to write programs in

PLANNER which suffer from combinatorial explosion, but on the other hand the language offers at least the possibility to write inference systems which do not. The prime idea here is that PLANNER (but not FOPC) allows the user to specify how a goal (i.e., theorem) is to be established. For example, we can tell it only to look to see if it is already in the data base, or we can specify that it should try to infer the fact from other facts. Even this already gives us some control over how much computation is done in search of an answer, but there are other options also, like "use the theorems named. . . .first, only use the theorems named. . . ., use any theorem with the following property first. . . ., use only theorems with the following property. . . . This in effect allows the user to include information which should help the system make the needed inferences without dying a horrible death by combinatorial explosion.

2. COPING WITH CONTRADICTION. This is no problem in PLANNER. Since one writes one's own theorems, one would have explicitly to write a theorem which derived anything from a contradiction before that would happen. Furthermore, PLANNER is very well suited to one particular kind of semi-contradiction which comes up all the time. We would like to be able to say that all people have two legs, without worrying about the few rare cases where this is not true. On the other hand if Bill only has one leg we should be able to note the fact. In PLANNER this is done by making "Bill has one leg" an assertion, and "All people have two legs" a theorem. Since the data base is checked first when trying to establish a goal, the system will find that Bill has one leg before it attempts to use the general theorem to show that he has two legs.
3. DATA VS. QUESTION DRIVEN INFERENCE. While it was difficult to accommodate data driven inference in FOPC, there is no problem in PLANNER. Indeed, antecedent theorems exactly cover this situation.

Answers to the five questions

Finally what does PLANNER say in response to the five questions we asked initially?

1. SEMANTIC REPRESENTATION. Essentially nothing. There is nothing in PLANNER which in any way restricts what we can put in an assertion. If we assume that some of our impressions of the world get translated into theorems, then PLANNER does say a little about the form of the theorem, namely that it is written in PLANNER, but that is hardly much of a restriction.
2. INFERENCE TRIGGERING. Again very little. As we have already noted, one of the benefits of PLANNER is that it does not make the

restrictions that FOPC makes in this area.

3. ORGANIZATION. Here PLANNER does tell us something, but in a very qualified way. PLANNER offers several built-in organizational features. The primary one is pattern directed invocation. The secondary one is the means given to choose which theorems will be used to satisfy a goal, or react to a new piece of information. However, it should be remembered that PLANNER is a programming language, and it is certainly possible to program in other organizations. Nevertheless, it would seem fair to criticise PLANNER if the built-in organizational features it offered were not the ones we needed. As we will see later, there is some reason to believe that this is the case.
4. INFERENCE MECHANISM. Like FOPC, PLANNER is primarily a theory of inference mechanism. Given a PLANNER theorem, there is no doubt about how to use it, hand it to the PLANNER interpreter and have it executed.
5. CONTENT. Again nothing.

A DEMON BASED SYSTEM

But what are the problems with PLANNER? Surely it must have some or else it would have been announced some time ago that all the problems of memory and knowledge have been solved. Well, we have seen one problem with PLANNER already in passing. Since it is a programming language, as well as a theory of fact use, it is possible to use PLANNER in so many ways that it is hard to pin down exactly what PLANNER commits one to. Now a programmer would call this flexibility, but a theorist must call it vagueness. So the first problem with PLANNER as a theory of knowledge and inference is that it is a vague theory.

Naturally, this first problem makes it quite difficult to find other problems. To do so we will have to stop looking at PLANNER itself, and instead look at some of the ways it has been used. In particular we will look at my work on children's stories. I will then suggest that some of the problems we see in my model are really problems with PLANNER itself.

The problem my model (or system) is concerned with is that which I have tacitly assumed throughout this paper; namely, given a piece of simple narration (or children's story) like (15) the system should be able to answer reasonable questions like (16)-(18).

(15) Janet needed some money. She got her piggybank (PB) and started to shake it. Finally some money came out.

Some typical questions would be:

(16) Why did Janet get the PB?
(17) Did Janet get the money?
(18) Why was the PB shaken?

The reader should be sufficiently attuned by this point to recognize that to answer these questions real world knowledge is needed. The model is solely concerned with the problem of using the necessary real world knowledge to make inferences and it does not explicitly consider problems of natural language *per se*. In particular it does not deal with syntax, and while it does deal to some degree with those problems at the boundary between syntax and inference (like determination of noun phrase reference) I will not consider these topics here.

We will assume that as the story comes into the program it is immediately translated into an internal representation which is convenient for doing inference. The internal representation of a sentence will be a group of MICRO-PLANNER assertions. The model will try to "fill in the blanks" of the story on a line by line basis. That is, as it goes along, it will try to make connections between events in the story (usually causal connections) and fill in missing facts which seem important.

Demons and base routines

Consider a fact like:

(19) If "it is (or will be) raining" and if "person P is outside" then "P will get wet."

We have an intuitive belief that (19) is a fact about "rain", rather than, say, a fact about "outside". Many things happen outside and getting wet is only one of them. On the other hand only a limited number of things happen when it rains.

We will embody this belief in our system by associating (19) with "rain" so that only when "rain" comes up in the story will we even consider using rule (19). We will say that rain is the "topic concept" of (19). To put this another way, when a concept is brought up in a story, the facts associated with it are *"made available"* for use in making inferences. (We will also say that the fact are *"put in"* or *"asserted"*.) So, if "circus", say, has never come up, the program will not be able to make inferences using those facts associated only with "circus".

Note however that we are not saying that "rain" has to be mentioned explicitly in the story before we can use (19). It is only necessary that there be a "rain" assertion put into the data base. Other parts of the story may provide facts which cause the program to assert that it is raining. For example:

(20) One afternoon Jack was outside playing ball with Bill. Bill looked up and noticed that the sky was getting dark. "I think we should stop" said Bill. "We will get wet if we keep playing".

Here, the sky's getting dark in the afternoon suggests that it is going to rain. If this is put into the data base it will be sufficient to bring in facts associated with "rain".

Also note that a topic concept need not be a single "key word". A fact may not become available to the system until a complex set of relations appears in the data base. A fact may be arbitrarily complex, and in particular may activate

other facts depending on the presence or absence of certain relations in the story.

Looking forward, looking back. When a fact is made available we might not have all the information needed to make use of the fact. Since we are making inferences as we go, if the necessary information comes in after the rule has been asserted we want to make the inference when the information comes in. So we might have:

(21) Jack was outside. It was raining.
(22) It was raining. Jack was outside.

In (21) there is no problem. When we introduce "rain" we have sufficient information to use (19) and infer that Jack is going to get wet. But in (22) we only learn that Jack is outside after we mentioned rain. If we want to use (19) we will need some way to have our fact "look forward" in the story. To do this we will represent facts by (MICRO-PLANNER) antecedent theorems, so a fact will have two parts, a pattern and body (an arbitrary program). We will execute the body of the fact only when an assertion is in the data base which matches the pattern. In (19) the pattern would be "someone outside". Then in (22) when we introduce (19) no assertion matches the pattern. But the next line creates a matching assertion, so the fact will be executed. We will say that a fact is *"looking foward"* when its topic concept appears before the assertion which matches the pattern. When the assertion which matches the pattern comes first we will say that the fact is *"looking backward"* (as in (21)). (This is a slight extension over MICRO-PLANNER antecedent theorems which can only look forward.)

We can see how important looking forward is with a few examples.

(23) Janet was thinking of getting Jack a ball for his birthday. When she told Penny, Penny said, "Don't do that. Jack has a ball." Here we interpreted the line "Jack has a ball" as meaning that he did not want another. The common sense knowledge is the fact that in many cases having an X means that one will not want another X. This piece of information would probably be filed under "things to consider when about to get something for somebody else". Naturally it was an earlier line which mentioned that Janet was thinking of getting Jack a ball.
(24) Bill offered to trade his pocket knife for Jack's dog Tip. Jack said "I will ask Janet. Tip is her dog too." The last line is interpreted as the reason Jack will ask Janet because of information about the relation between trading and ownership.
(25) "Janet wanted to get some money. She found her piggybank and started to shake it. She didn't hear anything." The last line means that there was nothing in the piggy-bank on the basis of facts about piggy-banks.

In each of these cases it is an earlier line which contains the information which is used to assign the interpretation. So in (23) there is nothing inherent in the line "Jack has a ball" which means "don't get him another". If there were, something in the line would also have to trigger a check for the following situations:

(26) Bill and Dick wanted to play baseball. When Jack came by Bill said "There is Jack. He has a ball."

(27) Tom asked his father if he would buy him a ball. "Jack has a ball," said Tom.

(28) Bill's ball of string was stuck in the tree. He asked Jane how he could get it out. Jane said "You should hit it with something. Here comes Jack. He has a ball."

I formulated facts as antecedent theorems because I was so impressed with the need to "look forward". However, rather than call the facts antecedent theorems, I call them *"demons"* since it is a shorter and more mnemonic name.

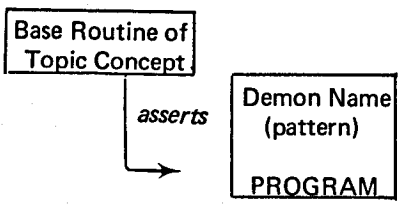
Specification and removal of demons. It should be emphasized that the model does not "learn" the information contained in the demons. This information is put in by the model maker. On the other hand, the demons are not specific to the story in the sense that they mention Jack, or "the red ball". Rather, they talk about "a person X" who at one point in the story could be Jack, at another, Bill. We will assume a mechanism for binding some of the variables of the demon ("specifying" the demon) at the time the demon is asserted.

We want demons to be active only while they are relevant to the story. A story may start by talking about getting a present for Jack, but ultimately revolve around the games played at his party. We will need some way to remove the "present getting" demons when they have outlived their usefulness. (An irrelevant but active demon not only wastes time and space, but can cause us to misinterpret a new line.) As a first approximation we will assume that a demon is declared irrelevant after a given number of lines have gone by.

Base routines. So far we have said that demons are asserted when the proper concept has been mentioned. But this implies that there is something attached to the concept telling us what demons should be put in.

If we look at a particular example, say (24), it is Bill's offer to trade which sets up the context for the rest of the fragment. I will assume that the information to do so is in the form of a program. Such routines, which are available to set up demons, will be called *"base routines"*.

The relation between base routines and demons can be represented as:



These base routines will be responsible for more than setting up demons. Suppose we are told that Jack had a ball, and Bill a top. Then Jack traded his ball to Bill for the top. One question we might ask is "Who now has the top?" Naturally since questions of "who has what" are important in understanding stories we will want to keep tabs on such information. In this particular case, it must again be the "trade" statement which tells us to switch possession of the objects. Every time a trade occurs we will want to exchange objects, so whenever we see "trade" we execute the "trade" base routine. Of course, the program can't be too simple-minded, since it must also handle "I will trade..." and perhaps even "Will you trade...?"

A good test as to whether a given fact should be part of a base routine or a demon is whether we need several lines to set it up or whether we can illustrate the fact by presenting a single line. (Naturally several lines could be made into one by putting "and's" between them, but this is dodging the point. I am only suggesting an intuitive test.) So we saw that "Jack has a ball" was not enough by itself to tell us that Jack does not want another ball. Hence this relation is embodied by a demon, not a base routine.

Bookkeeping and fact finders

Up to this point we have introduced two parts of the model, demons and base routines. In this section we will introduce the remaining two parts.

Updating and bookkeeping. Again let us consider the situation when Jack had a ball, Bill a top, and they traded. When we say that Bill now has the ball, it implies that Jack no longer does. That is to say, we must somehow remove the fact that Jack has the ball from the data base. Actually we don't want to remove it, since we may be asked "Who had the ball before Bill did?" Instead, we want to mark the assertion in some way to indicate that it has been updated. We will assume that there is a separate section, pretty much independent of the rest of the model, which is responsible for doing such updating. We will call this section "bookkeeping".

Fact finders. But even deciding that one statement updates another requires special knowledge. Suppose we have:

(29) Jack was in the house. Sometime later he was at the store.

If we ask "Is Jack in the house?" we want to answer "No, he is at the store." But how is bookkeeping going to figure this out? There is a simple rule which says that (<state> A B) updates (<state> A C) where C is not the same as B. So (AT JACK FARM) would update (AT JACK NEW-YORK). But in (29) we can't simply look for JACK AT <someplace which is not the store>, since he is IN the house. To make things even worse, we could have:

(30) Jack was in the house. Sometime later he was in the kitchen.

To solve this problem we will need:

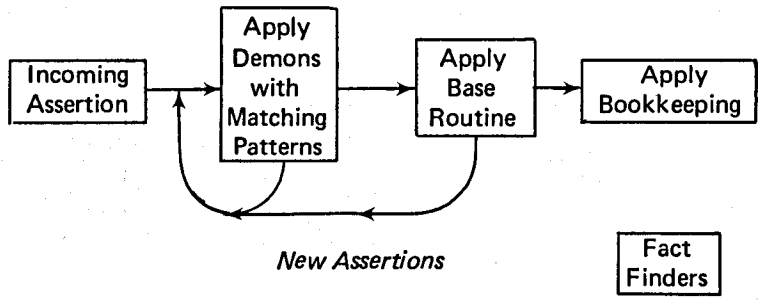
(31) To establish that PERSON is not at location LOC

Find out where PERSON is, call it X
If X = LOC, then theorem if false so return "NO".
If X is part of LOC then return "No".
If LOC is part of X, then try to find a different X
Else return "Yes".

In (29) the bookkeeping would try to prove that Jack is not at the store, and it would succeed by using (31) and the statement that Jack is in the house. Bookkeeper would then mark the earlier statement as updated. Theorems like (31) are called *"fact finders"*. Fact finders are represented as MICRO-PLANNER consequent theorems.

The basic idea behind fact finders is that they are used to establish facts which are comparatively unimportant, so that we do not want to assert them and hence have them in the data base. So in (29) we do not want to assert "Jack is not in the house" as well as "Jack is at the store". In the same way we will have a fact finder which is able to derive "<person> knows <fact>" by asking such questions as "was the <person> there when <fact> was mentioned or took place?" Again, this information is easily derivable, and not all that important, so there would seem to be no reason to include it explicitly in the data base.

Then, we can represent our model as:



The five questions

To summarize this, let us return to our five questions and see what answers this model gives. In answering them however we will have to be careful, since in a peculiar way the model just presented (and the ones we will consider henceforth) answers all of them. What I mean is that since this model was made into a computer program, and since one has to specify everything in a program, the model must have some answer to each of the questions. Yet, very little was said in the preceding pages about, say, semantic representation, and I would deny that I really said anything about semantic representation at all.

The way out of this peculiar contradiction is to make a distinction between a computer program and the theory a computer program embodies. There has been much written on the relation between programs and theories, and the last thing I want to do is add to this literature, but some distinction of this sort is necessary. The distinction I will make is quite simple. If person P writes an AI

program which performs task T, I will say that P's theory of T is those parts of the program which P considers significant. This definition has as a consequence that a single program could embody two different theories. This could happen, for example, if two people wrote a single program, but held different opinions about which portions of the program were significant.

So with this distinction in mind, what answers does my theory provide?

1. SEMANTIC REPRESENTATION. I have very little to say about semantic representation. Indeed the only statement which the preceding commits me to is that a semantic representation with *only* a small number of primitives is wrong. For example, in the preceding I used more specific concepts like "rain", "piggy-bank" and "present giving". However, to the best of my knowledge no one I know actually holds the contrary position so it is not clear that it is worth arguing the point. (Some people mistakenly believe that Schank holds this position, and Schank's writings often encourage this misapprehension, but from personal conversation I know that he does not.)
2. INFERENCE TRIGGERING. I am firmly committed to making non-problem occasioned inferences at read time, and for the reasons we went into above. This was the entire point of the demon apparatus.
3. ORGANIZATION. By and large the theory just presented is a theory of organization. In particular it states that given a particular assertion, the way we find those facts which we should use to make inferences from the assertion is to look in two places, first the base routine for assertions of that form, and second for any demons which happen to have been activated which are looking for assertions of that form. To put this slightly differently, the system presented states that the PLANNER mechanism of pattern directed invocation is the way facts are located.
4. INFERENCE MECHANISM. The theory assumes that the inference mechanism is MICRO-PLANNER demons (or the equivalent in some other language). A subsequent section will argue against this assumption.
5. CONTENT. As presented, the model said nothing about content in that it makes no claims about exactly what it is we know about piggy-banks, or anything else. One might use this model to make claims about what it is we know of these topics, but nothing in the presentation here does so.

Problems with the model

There are many places where one could find fault with the model just proposed, but I will pick one issue which seems to me important because it touches on the more general issue of the role of high level programming languages as

vehicles for theories of knowledge and inference.

Consider a fact like:

(32) Umbrellas are used to keep rain off one's head.

To fit such a fact into the model just presented, we would most naturally treat it as follows:

(33) Base routine which activates demon: Possibility of rain.

Pattern: Person gets umbrella.
Program: If person might be caught in rain, he got the umbrella to prevent getting wet.

This will work quite well for stories like:

(34) It looked like rain. Jack got his umbrella.

It would even be possible, using mechanisms in the model which were not explained in the previous section to handle a story like:

(35) As Jack was leaving the house, he heard on the radio that it might rain. He went to the closet.

If asked why he did this we would respond that he was probably getting an umbrella. (It would be also possible to answer that he was getting his raincoat, but this is not important since "raincoat" would also have information connecting it to rain.) The extra mechanism which is needed here is the ability to put together the "expectation" of getting an umbrella, with our knowledge of where umbrellas are normally kept to conclude that he is going to get his umbrella in spite of the fact that the word "umbrella" was never mentioned in (35).

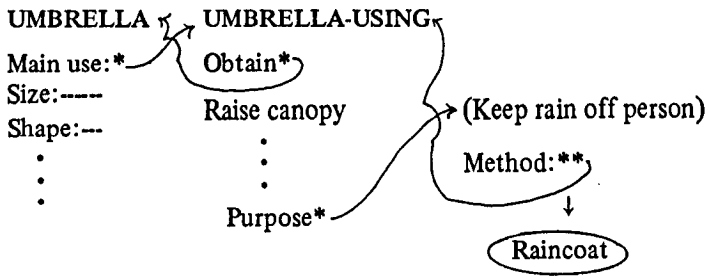
The trouble with this solution is that it would not account for the following story:

(36) Jack began to worry when he realized that everyone on the street was carrying an umbrella.

Question: What was Jack worrying about?
Answer: That it might rain, and he was without an umbrella.

While it is intuitively clear that a fact like (32) comes into play here, the formulation in (33) as vague as it is, is incapable of accounting for (36). The problem is that since (36) never mentioned rain, the demon expressed in (33) would never have been activated. To put this in terms of pointers, the fact in (33) only allows a pointer from "rain" to "umbrella", it does not allow a pointer from "umbrella" to "rain" and hence cannot be used to help us conclude in (36) that the problem is rain. Now it is not hard to come up with solutions to this problem. I can think of several alternatives, of which my favorite is something like:

(37)



What (37) says is that there is a heading under "umbrella" which indicates the use of umbrellas by pointing to a set of facts about umbrella usage. Umbrella usage naturally points right back at umbrella because to use an umbrella you must have one. It also notes that the purpose of using an umbrella is to keep rain off one, and keeping rain off one notes that one standard way to accomplish this end is to use an umbrella. Now with the possible exception of the facts listed under "umbrella-using" none of this looks very much like a program, or a demon. It is rather a complex set of pointers, which is just another way to say "data structure". Furthermore, I have implicitly argued elsewhere (Charniak, 1975) that even umbrella-using cannot be considered to be "program" but rather looks like "data".

Now all this argument is highly speculative. First, I am arguing that the proper representation for fact (32) looks more like data than program, and from that conclusion I wish to argue against PLANNER, and the "proceduralist" view in general. Both phases of the argument can be attacked. For one thing, it is conceivable that we could change demon (33) so that it could also be activated by "umbrella". This would not be easy since it would require that the program take into account under which circumstances it was activated, but it conceivably could be done. The response, of course, is the same one we gave concerning FOPC. When we have to subvert the natural inclinations of a system there is probably something wrong. (It is also true that there is no hard line between what might be considered program, and what data. One of the factors which helps determine whether we consider a particular structure as one or the other is to what degree the structure determines how it is to be used. By allowing programs to become usable in more than one way we are making them slightly less "programlike". One must watch for the possibility that in gradually modifying one's program to account for cases like (36) it becomes "data" without one's realizing it.)

Also note that by giving up the use of demons to represent individual facts like (32) we are implicitly giving up the mechanism of pattern directed invocation as our means of locating useful facts. That is, since we no longer represent (32) as a demon there is no longer the pattern-program distinction which makes the concept of pattern directed invocation meaningful.

But saying that (32) is better expressed in a data format like (37) than a demon format like (33) is not necessarily to say that knowledge should not be

expressed in PLANNER. As we noted earlier, PLANNER is a programming language and hence has a certain amount of flexibility. To consider only one possibility, we could agree that (32) should not be expressed as a distinct demon, while arguing instead that it is better expressed as part of a program which, say, includes all of the information expressed in (37), but unlike (37) is written in PLANNER. My own opinion is simply to repeat what by now has become an old refrain, I am suspicious when one has to manoeuvre around the problems inherent in a formalism.

FRAMES

There are other problems with the demon based scheme (see "A better looking supermarket frame," below). My reaction to them has been a shift to a different formalism. In particular my most recent work has been based on an idea suggested by Minsky (Minsky, 1974), the "frame". It is perhaps indicative of the convergence of ideas reflected in (or perhaps inspired by) Minsky's paper that the overall organization proposed here (although not the details) is quite similar to the independently developed "scripts" of (Schank and Abelson, 1975).

I take a frame to be a static data structure about one stereotyped topic, such as shopping at the supermarket, taking a bath, or piggy banks. Each frame is primarily made up of many statements about the frame topic, called *"frame statements"* (henceforth abbreviated to FS). These statements are expressed in a suitable semantic representation, although I will simply express them in ordinary English in this paper.

The primary mechanism of understanding a line of a story is to see it as instantiating one or more FS's. So, for example, a particular FS in the shopping at the supermarket frame would be:

(38) SHOPPER obtain use of BASKET

(SHOPPER, BASKET, and, in general, any part of an FS written in all capitals is a variable. These variables must be restricted so that SHOPPER is probably human, and certainly animate, while BASKET should only be bound to baskets, as opposed to, say, pockets.) This FS would be instantiated by the second line of story (39).

(39) Jack was going to get some things at the supermarket.
The basket he took was the last one left.

Here we assume that part of the second line will be represented by the story statement (SS):

(40) Jack1 obtain use of basket1

(Of course, really both (38) and (40) would be represented in some more

abstract internal representation.) Naturally, (40) would be an instantiation of (38), and this fact would be recorded with a special pointer from (40) to (38).

The supermarket frame will contain other FS's which refer to (38), such as:

(41) (38) usually occurs before (42)
(42) SHOPPER obtains PURCHASE-ITEMS

Any modification (like (41)) of a particular FS (like (38)) will be assumed true of all SS's which instantiate that FS (like (40)), unless there is evidence to the contrary. Hence, using (41) we could conclude that Jack has not yet finished his shopping in (39). Other modifications of (38) would tell us that Jack was probably already in the supermarket when he obtained the basket, and that he got the basket to use during shopping.

The variable SHOPPER in (38) also appears in (42), and in general a single variable will appear in many FS's. Hence the scope of these variables must be at least that of the frame in which they appear. When an SS instantiates an FS the variables in the FS will be bound. Naturally it is necessary to keep track of such bindings. For example, failure to do so would cause the system to fail to detect the oddness in (43) and (44).

(43) Jack went to the supermarket. He got a cart and started up and down the aisles. Bill took the goods to the checkout counter and left.
(44) Jack went to the supermarket to get a bag of potatoes. After paying for the milk he left.

It is probably a bad idea to actually change the frame to keep track of such bindings. Instead I assume that the frame remains pure, and that the variable bindings are recorded in a separate data structure called a *"frame image"* (abbreviated FI). For frames which describe some action, like our shopping at supermarket frame, we will create a separate FI for each instance of someone performing the action. So two different people shopping at the same time, or the same person shopping on two different occasions, would require two FI's to record those particulars which distinguish one instance of supermarket shopping from all others.

Much of this information will be stored in the variable bindings (shopper, purchase items, store, shopping cart used, etc.) However, the variable bindings do not exhaust the information we wish to store in the FI, for example it will probably prove necessary to have pointers from the FI to some, if not all, of the SS's which instantiate FS's of the frame in question. Of slightly more interest is that the FI of a frame describing an action will keep track of how far the activity has progressed. So, for example, we would find the following story odd:

(45) Jack drove to the supermarket. He got what he needed, and took it to his car. He then got a shopping cart.

We have already said that FS's are modified by time ordering statements, so to

note the oddity of (45) it is only necessary to have one ore more *progress pointers* in the FI to the most time-wise advanced FS yet mentioned in the story. Then when new statements are found in the story which instantiate FS's in the frame, the program will automatically check to see if these FS's are consistent with the current progress pointer(s). If so, the FI progress pointer(s) may be advanced to indicate the new state of progress. If not, as in (45), the oddity should be noted, and, if possible, the story teller questioned about the oddness of the time sequence.

I have not commented so far about how, given a new SS, we locate an FS which it instantiates. In general this is a difficult problem, and I will have little to say about it. Roughly speaking the problem falls into two parts. First, the system must recognize that a given frame is relevant to a particular story. I am assuming that the presence of a key concept in the story will trigger a given frame. (It should be clear however that this is much too simple minded. For example, the scene setting description of a city block as containing a supermarket, bank, tailors, shoe repair shop, etc., should probably not activate the frames for the activities normally done in each.) Secondly, given that one or more frames have been selected as relevant to the story, how does the program find the particular FS which is instantiated by a particular SS? Here I will assume that a list of current frames is kept and frames which have not been used recently are thrown away. To find the particular FS which is instantiated by the SS I will simply assume that all FS's of the recently used frames are checked for a match. A more sophisticated procedure would be to first check FS's which follow the progress pointer. Another improvement would be to have an index for each frame so that it would not be necessary to check all FS's of the frame. This would, in effect, make each frame into a local data base. If a frame has a sub-frame it too will be checked, although it will probably be necessary to put some limit on how deep one should look into sub-frames.

One final note before moving on to more detailed issues. This paper is concerned primarily with the use of frames in the comprehension of simple narration. However, it seems only reasonable to me to assume that whatever knowledge we have built up into frames was done, in large part, in order to get around in the world, rather than to read stories. I will assume then that the same knowledge structures should be usable for either task, and upon occasion I will make arguments that structuring a frame in a particular way will make it easier to perform actions based on the frame structured knowledge. It seems to me one of the great advantages of frames is that they seem capable of being used in multiple ways, something which is not obviously true, for example, of demons.

FRAMES AND SUB-FRAMES

In this and the next two sections we will take a closer look at the internal structure of a frame. In particular I will try to show that Minsky's notion that different frames will have "terminals" in common is applicable to the kind of frames I have in mind. Minsky's idea was that frames applied to problems of

vision would store information about what one was likely to see in a certain situation (e.g., a room) and from a certain vantage point (e.g., just having walked in the door). Upon changing vantage points one moved to different frames, but many of the "terminals" of the frame (e.g., right wall, center wall, lamp, etc.) would appear in both views, and hence in both frames. While Minsky used the term "terminal" when discussing scenario frames (his "terminals" very roughly correspond to my "frame statement") he never applied the idea of common terminals between frames to scenario frames. Nor is it clear that frames such as the ones discussed in this paper have anything directly corresponding to the sharing of a wall terminal between two room frames. However, I will try to show that Minsky's notion is useful in a somewhat different way.

Let us start by giving a naive outline of some FS's about supermarkets.

(46) a) Goal (SHOPPER own ITEMS)
b) SHOPPER be at SUPERMARKET
c) SHOPPER have use of BASKET
d) do for all ITEM ϵ ITEMS
e) SHOPPER at ITEM
f) BASKET at ITEM
g) ITEM in BASKET
h) end
i) SHOPPER at CHECKOUT COUNTER
j) BASKET at CHECKOUT COUNTER
k) SHOPPER pay for ITEMS
l) SHOPPER leave SUPERMARKET

I am assuming in (46) an implicit time ordering from top to bottom. In the actual frame this time ordering would be made explicitly. The reader might also notice that most of the FS's are states which must be achieved at some point in the course of the action. For reasons why I use states rather than actions to express what happens in an action sequence, see (Charniak, 1975).

There are countless things missing or wrong with (46), but I wish to concentrate on only one of them, the relation between shopping and using a shopping cart. It should be obvious that one can do shopping without using a cart, although (46) would make it seem that cart usage is an indispensable part of shopping. I will suggest that a good way to gain this flexibility in the supermarket frame is to have a separate cart frame which shares information with the supermarket frame, by having, in effect, some FS's common between the two.

One way to account for the ability to shop with or without a cart (and to understand stories about same) would be to have a second frame for shopping without a cart. However, not only is the idea unsatisfying, since it would require the duplication of the many facts the two activities have in common, but it would also lead to problems in the comprehension of certain types of stories. For example, one could easily imagine a story which starts out with Jack using a cart, the wheel of the cart sticking, and rather than going to get a second cart Jack finishes his shopping without a cart. It seems *a priori* that such combina-

tions would be extraordinarily hard to account for with two completely separate frames for the two forms of supermarket shopping. (Alternatively, it is common practice in crowded situations to park one's cart in a general vicinity of several items and pick up the individual items without the cart, only to bring them all to the cart at a later point and resume shopping with the cart.)

One possibility for a single frame which handles both kinds of supermarket shopping is:

(47) a) Goal: SHOPPER owns PURCHASE-ITEMS
b) SHOPPER decide if to use a basket. If so, set CART to T
c) If CART then SHOPPER obtain BASKET
d) SHOPPER obtain PURCHASE-ITEMS
method
e) → Do for all ITEM ϵ PURCHASE-ITEMS
f) SHOPPER decide on next ITEM
g) SHOPPER at ITEM
h) If CART then BASKET also at ITEM
i) SHOPPER hold ITEM
j) If CART then ITEM in BASKET
k) End
l) SHOPPER at CHECK-OUT COUNTER
m) if CART then BASKET at CHECK-OUT COUNTER
n) SHOPPER pay for PURCHASE-ITEMS
o) SHOPPER leave SUPERMARKET

Now one problem with (47) is that to my eye the constant repetitions of "If CART then . . ." give it a rather *ad hoc* appearance. And while it was this ugliness which gave me the initial impetus to find a better representation, there are more concrete problems with (47). One major one is the lack of any information about why these various actions are to be performed and why in this particular order. This lack comes out most strongly when we consider stories where something goes slightly wrong in the course of the shopping. For example:

(48) Jack was shopping at the supermarket. After getting a few things he returned to his cart only to find that some prankster had taken everything out of it and put the things on the floor. After putting the things in the cart, Jack finished his shopping, but was not able to find out who had done it.

Question: Why did Jack put the groceries which were on the floor into his cart?

The question here is so simple that one might be at a loss to know what sort of answer is desired, but one certainly knows that the items were put back in the cart so Jack could continue his shopping. The point here is that frame (47) does

not give any explanation for Jack's action in (48). Naturally we would not expect any supermarket frame to explicitly take into account strange situations like (48), but this is not necessary to be able to answer the question in (48). All that is needed is an understanding that the purpose of a cart is to transport goods from place to place in the supermarket and that to do this the goods must be in the cart. Hence, in this situation, if, as is most likely, Jack still wants to transport the goods elsewhere he should once again put the items in the cart. But (47) does not give this information. It tells us to put the items in the cart, but not why or for how long.

I should point out that if (48) seems like an extraordinarily odd story on which to base any conclusions, there are much more normal ones which make the same point. For example:

(49) Jack was shopping at the supermarket. After getting a few items the wheel of his cart stuck. He got a second cart and finished his shopping.

Question: Why did Jack transfer his groceries to a second cart?

The story does not say that Jack transferred his groceries, and to infer that he did require essentially the same reasoning process required to understand why Jack put the groceries back in the cart in (48).

To handle stories like (48) and (49) we must therefore put two pieces of information into (47) which are not there at present. First, that in those cases like lines (47 h) and (47 m) where we have the basket going along with the person, the reason is to keep the previously collected items with one. Second, that to carry something with a cart requires that it be in the cart. We will indicate the first of these by:

(50) SHOPPER at ITEM
side condition - DONE at ITEM also
| method - suggested
↳ cart-carry (SHOPPER, BASKET, DONE, ITEM)

Here DONE is a list of those items already collected. Cart-carry is a frame (and hence a sub-frame of the supermarket frame) describing the use of a cart for carrying things. (I am introducing a bit of terminology (method-suggested) from (Charniak, 1975); I will assume that it is reasonably self-explanatory. Consult (Charniak, 1975) for some explanation and justification.) The FS's in (50) replace lines (47 g) and (47 h) and differ from them in two respects. First (50) formulates the goal in a manner neutral with respect to using a basket or not, with only a suggestion that a basket be used. This is obviously necessary if we are to handle stories where the person does not use a basket. Secondly, it assumes the existence of a separate cart-carry frame in which we store information about using carts to carry things. We will see other advantages of this move later, but at the moment we can at least note that if one were to ask "Why does

Jack use a basket?" two answers (at least) would be possible–"to do shopping", or "to carry his groceries". (50) allows for both of these answers, wheras (47 g) and (47 h) only allow for the former, since there is no separate "carry" level.

The second piece of information we needed to handle stories (48) and (49) was that to carry something with a basket it is necessary that the thing be in the basket. The most natural place to put such information would be in our newly created cart-carry frame where it would be some sort of a pre-requisite (or more precisely a "strict" (as opposed, for example, to "suggested") substrate to use the terminology of (Charniak, 1975). By creating the cart-carry frame and locating information about the action within it, we also circumvent the need to duplicate this information elsewhere. For example, an expanded supermarket frame would include the fact that in some circumstances it is permitted (and suggested) that one uses one's basket to take the groceries to one's car. By stating this as suggesting cart-carry (SHOPPER, BASKET, PURCHASE-ITEMS, CAR) we no longer need an instruction to put the groceries into the basket again before setting out.

SHARING FRAME-STATEMENTS BETWEEN FRAMES

So far, then, I have argued that frames must be able to reference sub-frames, and in particular, the supermarket frame needs some sub-frame like cart-carry. There is nothing exceedingly strange in this, but the next step will perhaps be a bit more interesting. Here I will suggest that some of the frame statements in our supermarket frame be shared with the cart-carry frame. To see the reasons for this, let us start by noting that the failure to allow for common FS's will lead to some curious redundancies in our frame. One of these occurs in the DO loop of (47) which handles the collection of the PURCHASE-ITEMS. With out latest changes, this portion of (47) (lines (e) through (k)) looks like:

(51) a) Do for all ITEM ϵ PURCHASE-ITEMS
b) SHOPPER choose next ITEM ϵ PURCHASE-ITEMS - DONE
c) SHOPPER at ITEM
d) side-condition DONE at ITEM also
e) | method - suggested
f) ↳cart-carry (SHOPPER, BASKET, DONE, ITEM)
g) SHOPPER hold ITEM
h) If CART then ITEM in BASKET
i) DONE ← DONE + ITEM
j) End

The redundancy is this: we have already stated that cart-carry has a strict sub-state that the things to be carried must be in the basket. But in (51) we further specify in line (h) that ITEM is to be in BASKET, which is simply a special case.

Once again my initial reason for being concerned with this is that I find such redundancy unappealing, but again there seem to be more solid reasons for doing away with it. In particular consider the following story:

(52) Jack was doing some shopping at the supermarket using a shopping cart. The last thing he got was a package of gum, which he picked up right at the check-out counter. Jack put the gum and everything else down in the counter.

Question: Why didn't Jack put the gum in his basket?

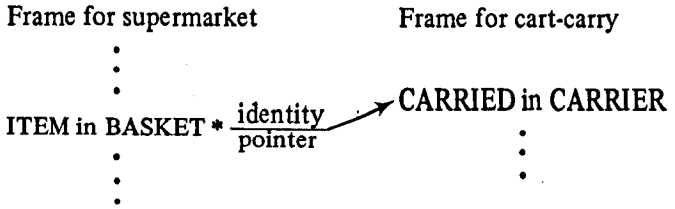
Again the question seems silly, but again we all know that there would be no reason to put the gum into the cart simply because Jack already has everything at the check-out counter. But (51) as stated does not allow us to make this inference. It simply says to put each ITEM into the BASKET and since no reasons are given there is no way to see that in the particular case of the gum in (52) there is no reason to put the gum into the basket.

My solution to this problem is to see line (51 h) as shared between the supermarket frame and the cart-carry frame. Looked at in this light, the reason for obeying (51 h) is then the same as the reason for having items in the basket as stated in the cart-carry frame—if you wish to use the cart to carry an item it must be in the cart. Since Jack has no reason for carrying the gum in the cart he has no reason to put it in the cart. (To be a bit more precise, Jack might have a reason to put the gum in the cart, namely using the cart to carry the groceries to his car afterwards, but this only occurs after putting the groceries on the check-out counter, hence does not count as a reason for doing it at the particular point in time we are discussing.)

Now when I suggest that the two frames share an FS, I do not mean that the one FS physically appears in both frames, although this would be perfectly possible in a list processing language like LISP. There are, however, several good reasons for not implementing FS sharing by physical identity. For one thing it would mean that different frames would have to have the same variables, which at the very least would create a major debugging problem. From a theoretical point of view it seems likely that such an attempt will run into trouble because two or more statements in one frame will share the same FS in a second frame. If the two FS's in frame one have different variables there would be no way for the FS in frame two to be identical to both, and hence could not be physically the same. (However, I do not have a clear-cut example of this happening.)

So I will not assume that FS (51 h) is physically identical to the corresponding FS in cart-carry, but rather that there is a pointer from (51 h) to the FS in cart-carry which says that (51 h) should be considered to be the same FS. That is, we would have an arrangement somewhat like:

(53)



Here I have created two new variables for the cart-carry frame, CARRIED which specifies what is carried, and CARRIER which is the cart used to carry. Naturally, to actually use the cart-carry information for understanding the ITEM and BASKET line of the supermarket frame it will be necessary to see that ITEM corresponds to CARRIED, etc. (It may also be necessary to see that SHOPPER in supermarket corresponds to the variable for the actor in cart-carry, and this would require more formalism, but I am not sure it is necessary.) Finally, note that there seems to be no reason to have a corresponding pointer from the FS in cart-carry to ITEM in BASKET, since there is no need to know about supermarkets in order to use a basket.

Now if all of this seems eminently reasonable to you, feel free to skip the next section. But for those of you to whom this seems a strange sort of data structure, what follows is an attempt to justify it by considering one alternative and showing how the shared FS proposal is superior.

Beyond the "ugliness" of the redundancy, the only argument we gave for replacing (51 h) with (53) was the story (52) where Jack did not put the gum into the shopping cart. One way to solve both the redundancy, and the problem spotlighted by (52), would be to simply remove (51 h) from the supermarket frame. This clearly solves the redundancy problem, and it also solves (52) since now the only FS to the effect that the goods should be in the basket appears in cart-carry, and since Jack has no reason for carrying the gum in the cart he has no reason to put the gum into the cart.

The argument against this possibility must start from the recognition that it has some counter-intuitive properties. Intuitively one sees putting ITEM into BASKET as the last state of collecting ITEM. With this new solution this is no longer the case. Instead, putting ITEM into BASKET is a result of wanting to move to the next ITEM. (To distinguish we will call the item one has just obtained $ITEM_n$ and the item one is going to obtain next $ITEM_{n+1}$.) Hence, putting $ITEM_n$ into the basket is not the last thing of the N'th cycle, but one of the first of the N+1'st. This is against my intuition and makes me immediately suspect it.

Furthermore, this counter intuitiveness seems to have more substantive implications. For example:

(54) Jack was shopping at the supermarket. After getting a basket he went to the milk counter and picked up a carton of milk. He then thought about what to get next.

Question: Did Jack put the milk in the basket?
Answer: I would assume so.

The shared FS model would allow for this answer since deciding on $ITEM_{n+1}$ occurs after putting $ITEM_n$ into the basket. But by deleting (51 h) we remove this information so there would be no way to answer the question other than "I don't know". (Of course, "I don't know" is also an acceptable answer, but our model must allow for the various alternative answers people can give.)

A second argument against deleting (51 h) comes from using our supermarket frame in actually doing shopping. By deleting (51 h) we would be saying in effect that one puts $ITEM_n$ into the basket when checking to see that all of DONE is in the basket when going to get $ITEM_{n+1}$. Computationally it seems horribly inefficient to bother to check on all of DONE each time (and in fact I am sure that people do not do it).

Finally by deleting (51 h) we make it difficult to account for "mistakes" that people make. For example, suppose we had the variation of (52) where Jack does put the gum into his cart only to immediately take it out again. We then ask the question:

(55) Question: Why did Jack put the gum into the cart, since he only had to immediately take it out again?
Answer: Well, I suppose one normally puts things into the cart immediately after picking them up.

Such considerations lead me to reject the alternative of deleting (51 h).

A BETTER LOOKING SUPERMARKET FRAME

So far I have argued for the shared FS model primarily on the basis of its ability to handle stories where mistakes occurred, but it is also the case that it allows us to solve one of the "aesthetic" problems mentioned earlier, namely the constant repetition of "IF CART then . . ." in (47). What we find is that all occurrences of this phrase in (47) can be replaced by either an explicit call to cart-carry (as in (50)) or an identity pointer to an FS in cart-carry, as in (53). To give another example of this, consider line (47 c), repeated here:

(47) c) If CART then SHOPPER obtain BASKET

This is clearly another example of a pre-requisite of cart-carry, and hence should be considered shared with cart-carry in the same way that ITEM in BASKET is shared. Furthermore, we can now remove the "If CART then" portion of (47 c) by assuming the eminently reasonable convention that a shared node in frame-1 which has a pointer to frame-2 is only applicable to the action in frame-1 if frame-2 is activated in the sense that we have created a frame image for frame-2. When performing the action in real life this means that upon deciding to use a cart, one sets up a cart-carry image and this in turn makes the various FS's in the supermarket frame dealing with carts relevant to one's activities. While reading a story the general rule will be that any SS instantiating an FS which is shared with cart-carry will be sufficient to create an FI for cart-carry. With this convention (47 c) becomes:

(56) SHOPPER obtain BASKET * $\xrightarrow[\text{pointer to cart-carry}]{\text{identity}}$

Furthermore, since I have argued that these pointers are needed on independent

grounds, we have received this simplification for free.

With both this simplification, and the use of cart-carry, our supermarket frame now looks like:

(57)
a) Goal: SHOPPER owns PURCHASE-ITEMS
b) SHOPPER decide if to use basket, if so set up cart-carry FI
c) SHOPPER obtain BASKET *cart-carry
d) SHOPPER obtain PURCHASE-ITEMS
e) | method - suggested
f) └→ Do for all ITEM ∈ PURCHASE-ITEMS
g) SHOPPER choose ITEM ∈ PURCHASE-ITEMS - DONE
h) SHOPPER at ITEM
i) side-condition DONE at ITEM also
j) | method - suggested
k) └→cart-carry (SHOPPER, BASKET, DONE, ITEM)
l) SHOPPER hold ITEM
m) ITEM in BASKET *cart-carry
n) DONE ← DONE + ITEM
o) End
p) SHOPPER at CHECK-OUT-COUNTER
q) side-condition PURCHASE-ITEMS at CHECK-OUT-COUNTER also
r) | method - suggested
s) └→cart-carry (SHOPPER, BASKET, PURCHASE-ITEMS, CHECK-OUT-COUNTER)
t) SHOPPER pay for PURCHASE-ITEMS
u) SHOPPER leave SUPERMARKET

I have here adopted the convention of indicating an identity pointer to another frame by a "*" followed by the name of the second frame.

In spite of the "simplification" introduced, (57) is considerably longer than (47). But on the other hand, (57) shows much more of the structure of shopping at supermarkets than does (47). To point out only one way where (57) is superior, it, but not (47), states that it is necessary to get one's groceries to the checkout counter, whether or not one uses a cart. Of course, (57) still does not contain more than a fraction of our knowledge of supermarkets, and in fact many of its particulars are clearly wrong, but it's a start.

INFERENCE ON FRAME BASED KNOWLEDGE

So far I have been discussing the organization of knowledge and have suggested that a large portion of it is stored in frames which connect up to other frames by either sub-frame relations or identity pointers. But a quick look back will reveal that at the same time several issues of inference have crept in. For example near the very beginning we stated that if an SS instantiates an FS then any modification of the FS within the frame is true of the SS also unless there is explicit information to the contrary. This, you will remember, allowed the system to conclude that when Jack got the shopping cart he was most likely already at the supermarket, but had yet to begin the actual act of collecting the groceries. Or again, when I mentioned that an FI had one or more progress

pointers, I implicitly assumed that one could infer what actions had already taken place using the time sequence information in the frame plus the progress pointer in the FI.

This is, of course, as it should be. One cannot, or at least should not, discuss structure independently of use and the use of frames is to allow us to make inferences about the stories we read. Nevertheless, there remain many issues of inference left untouched by the previous discussion and I will cover one or two of them here.

One problem is the inference triggering question of which inferences should be made at read time. It is my impression that the frame-like system just outlined has several nice properties in this regard.

For one thing, many inferences which would have to be made in a demon system need not be made in the system we have just outlined. To take again the example of Jack getting a shopping cart, I pointed out that the supermarket frame allowed the system to make several inferences about the statement, like why he did it, and that he had yet to start the shopping, but I left it vague as to whether these inferences should actually be made at read time, or only if a question was asked. In fact, there seems to be little reason for actually making most of these inferences at read time. Since the SS will have a pointer to the FS it instantiates, we may assume that a standard tactic for answering questions about a particular SS, like why the action was performed, or where, or when, would be to look in the frame for the answer.* To put this slightly differently, when an SS instantiates an FS it is not necessary to put into the data base instantiations of all the modifications of the FS.

Looked at in this light, it is interesting to ask under what circumstances one *would* want to instantiate an FS and put it in the data base. This is, after all, a large class of potential inferences and some restrictions on them would be at least a start on the problem of which inferences need be made at read time. The best answer I currently have to this question is summed up in the following rule:

(58) The Dual Usage Rule: If X is an FS in an active frame (one which has an FI) then X will only appear instantiated in the data base if it has two purposes.

Some typical purposes are:

a) appearing in an active frame
b) appearing in the semantic representation of the text
c) updating older statements in the data base

(this list will surely be expanded)

*I might point out in passing that the use of the modifying information in such a manner is a major reason why frames as I describe them look like "data" rather than "program". Traditionally programs have the property that they are not meaningful "locally" whereas we want to be able to use a modifying FS to answer a question about an SS without going through the entire frame. Hence my description of frames earlier as "a static data structure".

So the typical example of an FS which appears instantiated in the data base is one which appeared in the semantic representation of the text and was found to instantiate some FS. A more interesting case is exhibited by:

(59) Jack was going to Bill's birthday party. He thought Bill would like a kite. Jack then went to the store.

In (59) we should expect Jack to be buying a kite at the store. But consider the statement

(60) Jack1 own kite1

On one hand we can only predict that this is likely to occur using information from the birthday frame (perhaps in conjunction with the give sub-frame). On the other hand, this statement serves as the goal statement in the store frame. To enable (60) to act as the link between frames in this fashion it must appear explicitly in the data base with frame pointers to two different frame statements in two different frames. That is to say, (60) appears in two active frames, and hence is a justified instantiation.

It seems to me that (58) is a fairly strong rule, and it will be interesting to see if it can be maintained. Of course, given that the list of uses can be expanded, it is not completely clear what would serve as a counter example to the rule, but it seems to me that the general intent should be clear. For example, (58) would prohibit many of the inferences made by Rieger's system (Rieger, 1974). To give only one example, given statement (61) Rieger's system would infer statements like (62)-(65) none of which would qualify by the standard set up by (58).

(61) John told Mary that Bill wants a book
(62) John believes that Bill wants a book
(63) Mary now knows that Bill wants a book (assuming that the representation of "tell" is something like "cause to know by word of mouth")
(64) Bill might get himself a book
(65) John may want Mary to give Bill a book

(This is one of Rieger's own examples.)

COMPARISON WITH THE DEMON APPROACH

To get some perspective on frames as presented here let me compare them to the demon based model.

We already saw how demons (but not frames) had problems with our fact about umbrella using. To make further comparisons it is necessary to note the correspondence between frame statements and demons. This analogy works because frame statements accomplish precisely what demons were designed to accomplish—assign significance to a line due to the context it is in. So we notice how lines like (66)-(69) in the context of supermarket will instantiate FS's in the

supermarket frame and hence (66)-(69) will be given more significance than they have out of context.

(66) Jack got a cart.
(67) Jack picked up a carton of milk.
(68) Jack walked further down the aisle.
(69) Jack walked to the front of the store. He put the groceries on the counter.

What is interesting in this comparison is that one demon usually has a minimum of three or four statements, whereas obviously a single FS is only one statement. FS's seem then to have a considerable conciseness to them, at least when compared to demons. The reason for this is not hard to see. Demons, being independent facts, must bind their own variables, and much of the size of a demon is due to checks to make sure that the variable bindings are correct (e.g., BASKET must be a basket, and not a carton of milk). These same things must be checked in a frame, but since the scope of the variables is the entire frame, rather than a single FS, the overhead, so to speak, is shared. Furthermore, the inferences about a given FS are stored implicitly in the structure of the frame, whereas they had to be stated explicitly in the demon. So a second advantage of the frames approach over demons is the conceptual economy one obtains in the expression of facts.

The analogy between FS's and demons also points to a third way in which the frames approach seems superior. One problem which bothers many people (including myself) about the demon approach is that it seemingly calls for large numbers of demons to be activated every time a given topic is mentioned in the story, although it is unlikely that more than a small fraction of the demons will ever be used. There are two possible reasons why people feel this is a problem. One is that so many active demons might make it hard to locate those demons which really *should* apply. Using frames does not help with this problem since there will be equal numbers of FS's.

To see the second reason why activating large numbers of demons is problematic, note that if it took no time at all to set up a demon, setting up many of them would seem less bad. But of course it does take time to set up a demon, and it becomes a problem to justify this computation in light of the unlikeliness of the demon every being used. Frames do offer a potential solution to this second problem because with frames, rather than "supermarket" activating many demons, we need only create a frame image for one frame (i.e., supermarket). This would take much less time, and hence would be better, but it should be noted that we pay a price. In particular most of the work involved in setting up a demon is to index our storage of active demons so that retrieving the ones needed will be reasonably easy. By comparison looking through frames to find matching FS's promises to be a time consuming task unless we do something similar. This is what I meant earlier when I said that perhaps each frame would have its own index to its contents. On the other hand, the approach

presented here allows one to trade more time for locating an FS in return for less time to set up a new topic (frame), and the spectre of all those never-to-be-used demons makes me inclined to accept this trade.

Finally, the frames presented here have no problem handling time relations between FS's as we saw earlier in the paper. The same cannot be said of demons. We saw earlier how we might use a progress pointer to allow the program to notice actions which were out of sequence. What could be the equivalent in the demon model of the progress pointer? For one thing, where would such a pointer be stored? Short of giving every demon a pointer to the progress pointer, an inelegant solution at best, it is not clear what one could do. Furthermore, where would the time ordering information be stored? Notice that time ordering information is much more complex than a simple string, or even lattice which indicates the time orderings of actions. For example some time orderings are "strict" in the sense that one cannot possibly do things any other way, while others are "suggested" in the sense that it is a good idea to do the actions in a given order, but possible to do them some other way, while yet others are "regulatory" in the sense that it is possible, but illegal to do the actions in the opposite order (Charniak, 1975a). In the frames model one can store time ordering statements in the frame along with the rest. It is by no means obvious what to do in the demon model. This is not to say that one could not do it, but rather that having done it one would be left with something of little resemblance to the original demon model, and even less aesthetic appeal.

REFERENCES

Charniak, E. (1972) *Toward a model of children's story comprehension.* AI-TR266, MIT Artificial Intelligence Laboratory.

Charniak, E. (1975) *A partial taxonomy of knowledge about actions.* Paper submitted to IJCAI4.

Hewitt, C. (1969) *Planner: A language for proving theorems in robots.* (Eds. D. Walker and L. Norton) *Proc. IJCAI.*

Minsky, M. (1974) *A framework for representing knowledge.* AI Memo 306, MIT Artificial Intelligence Laboratory.

Raphael, B. (1968) Sir: A computer program for semantic information retrieval. (Ed. M. Minsky) *Semantic information processing.* Cambridge Massachusetts: M.I.T. Press.

Rieger, C.J. (1974) *Conceptual memory.* Unpublished Ph.D. Thesis, Stanford University.

Schank, R.C. (1973) *The fourteen primitive actions and their inferences.* AI Memo 183, Stanford Artificial Intelligence Laboratory.

Schank, R.C. and Abelson, R.P. (1975) *Scripts, plans, and knowledge.* Paper submitted to IJCAI4.

Sussman, G., Winograd, T., and Charniak, E. (1971) *Micro-planner reference manual.* AI Memo 203A, MIT Artificial Intelligence Laboratory.

Wilks, Y.A. (1975) *Notes for course "parsing english".* Lecture notes for the Tutorial on Computational Semantics, Institute for Semantic and Cognitive Studies.

27

Representation and Understanding of Text

Roger Schank

Department of Computer Science
Yale University

INTRODUCTION

How can we get a computer to understand natural language? Our view of the problem has progressed over the years to a point where an answer to that question today would look quite different from one given ten or even five years ago.

Originally, researchers felt that the most relevant issue was syntax. Later, most people agreed that semantics was the most relevant field of study (although few would have agreed on what semantics was). Five years ago, or so, our research was concentrated on finding an adequate meaning representation for sentences. We then began to run into the problems of inference. Even if we could analyze sentences for their meaning, we did not know exactly what we had to do with those meaning structures that we got as output. The answer turned out to be: find out more about what is true about what was said, and tie it together with other things you know or have just heard. In trying to do this we ran into the problem of organizing knowledge of the world. This is about where we are now.

In this paper, we shall run through the issues mentioned above (except for syntax which we never considered very relevant). First we shall discuss the basics of Conceptual Dependency representation. Then we shall discuss the primitive actions which are the basis of any hookup between analysis and inference. We shall then talk about the problems of connecting up causal relations as a way of getting at inference problems. We shall give a glimpse of the MARGIE program and then go on to discuss the theory of scripts and plans and how world knowledge considerations affect representation issues.

CONCEPTUAL DEPENDENCY

The first step in computer understanding of natural language must be the analysis of sentence into a representation of their meaning. We have required of the meaning structures that we use that they be unambiguous representations of

the meaning of an input utterance. We have required of those representations that they be unique. That is, the meaning representations of any two utterances which can be said to convey the same meaning should be identical.

Thus, we have concerned ourselves with the creation of conceptual structures, and the predictions and inferences that are possible given a formally defined conceptual structure.

Conceptual Dependency structures are intended to be a language-free unambiguous representation of the meaning of an utterance. Initially, however, the structures we used bore a great deal more similarity to the surface properties of English than we now believe should exist in such structures. Subsequently, we began looking for common concepts that could be used for representing the meaning of English sentences, that would facilitate paraphrase by the conceptual structures without losing information. The concept 'trans' was introduced (Schank, Tesler and Weber, 1970) as a generic concept into which words such as 'give' and 'take' could be mapped, such that by specifying attributed of the cases of 'trans' no information would be lost. (For example, 'trans' where the actor and recipient are the same is realized as the verb 'take', whereas, where the actor and donor part of the recipient case are the same, the verb is 'give'.) Such generic concepts simplified the conceptual networks, making them more useful. Furthermore, it became apparent that the linguist's problem of the representation of such concepts as 'buy' and 'sell' became solvable. Semanticists such as Katz (Katz, 1967) have argued that while these concepts seem close enough, it would be arbitrary to choose one as the basic form of the other, so the correct thing to do must be to write formal rules translating structures using 'buy' into structures using 'sell' when this is deemed necessary. Instead of doing this, we made the suggestion (Schank, 1972) that using 'trans' one could map 'buy' into 'trans money causes trans object' and 'sell' into 'trans object causes trans money.' Such a representation eliminates the 'which is more primitive than the other' problem and instead relates the two events that actually occurred.

The naturalness of the concept 'trans' led us to consider whether there might be more of these generic concepts around. Thus we began a search for primitive concepts that can be used as the basis of conceptual structures.

We use what is basically an actor-action-object framework that includes cases of the actions. That is, any action that we posit must be an actual action that can be performed on some object by an actor. Nothing else qualifies as an action and thus as a basic ACT primitive. The only actors that are allowed in this schema are animate. Thus, an action is something that is done by an actor to an object. (The exception to this rule regards natural forces and machines which do not really alter our concept of action.)

Actors, actions and objects in our conceptual schema must correspond to real world actors, actions, and objects. To illustrate what is meant by this consider the verb 'hurt' as used in 'Mary hurt John'. To treat this sentence conceptually as (actor: Mary; action: hurt; object: John) violates the rule that conceptual actions must correspond to real world actions. 'Hurt' here is a resultant state of

John. It does not refer to any action that actually occurred, but rather to the result of the action that actually occurred. Furthermore, the action that can be said to have caused this 'hurt' is unknown. In order to represent, in our conceptual structure, an accurate picture of what is going on here the following conceptual relationships must be accounted for: Mary did something; John was hurt; the action caused the resultant state. In conceptual dependency representation, actor-action complexes are indicated by ⟺, denoting a mutual dependency between actor and action; object-state complexes are indicated by ⟺ denoting a predication of an attribute of an object or by ⟸ denoting a change of state in the object. Causal relationships are indicated by ⇑ between the causer action and the caused action, denoting a temporal dependency. Causal arrows may only exist between two-way dependencies (⟺, ⟺ or ⟸). That is to say, only events or states can cause events or states.

Thus our representation for this sentence is:

Mary ⟺ do
⇑
John ⟺ hurt

The dummy 'do' represents an unknown action. ('Hurt' is ambiguous between mental hurt ($hurt_{MENT}$) and physical hurt ($hurt_{PHYS}$).)

Conceptual dependency representation, then, seeks to depict the actual conceptual relationships that are implicit within a natural language utterance.

Actions, in conceptual dependency, are things that are done to objects. Actions sometimes have directions (either through space or between humans), and always have means (instruments). These things are called the conceptual cases of an action. Unlike syntactic cases, (as posited by Fillmore [Fillmore, 1968] for example) conceptual cases are part of a given action and therefore are always present whenever that action is present. Thus, if an action takes an object, whether or not that object was mentioned it is considered to be present conceptually. If the particular instance of that object was not stated and is not inferrable, then an empty object slot is retained.

The conceptual cases are: OBJECTIVE; RECIPIENT; DIRECTIVE; and INSTRUMENTAL. Using the notion of 'trans' mentioned above we can deal with the sentence:

John gave Mary a book.

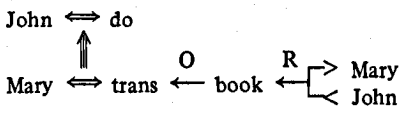
as follows:

John ⟺ trans ←(O)— book ←(R)— to → Mary
from

The symbol ←(O)— denotes 'object of the ACT' and the symbol ←(R)— to → / from denotes

'recipient of the object,' with the recipient of the object in the 'to' part, and 'donor of the object' in the 'from' part.

Actually, this analysis is not quite correct for this sentence since the sentence is conceptually ambiguous. The conceptual diagram above is correct for one sense of the sentence but it is possible that the transition was not done physically by John. Rather, John could have said 'you can have the book' and Mary would have taken it herself. Since we don't know what specifically John may have done we represent this sense as:

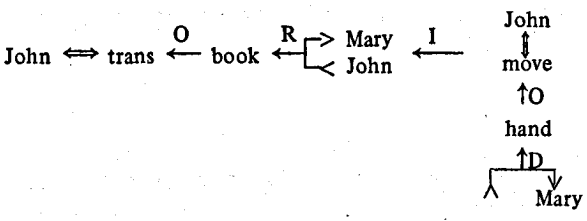


Either of these two structures may have been the intended one, but we assume unless given information to the contrary that the first is correct.

Suppose the sentence had been:

John gave Mary a book by handing it to her.

Here, the sentence is disambiguated by the 'by clause'. All actions require an instrument that is itself another actor-action-object complex (called a conceptualization). When the action in the main conceptualization is known; it is possible to delimit the set of possible instrumental actions. For 'trans' the ACT that is most often the instrument is 'move'. 'Move' represents the physical motion of a bodypart (which may be holding an object) by an actor, together with the direction that that action takes. The conceptual analysis then is:



The instrumental case is indicated by $\overset{I}{\leftarrow}$, and the conceptualization that is the instrument is dependent upon (written perpendicular to) the main conceptualization. The directive case (indicated by $\overset{D}{\leftarrow}$) shows the physical direction of the action. Thus 'the book was moved towards Mary'. (It is necessary to indicate here that the hand is holding the book also, but we shall not enter into that here.)

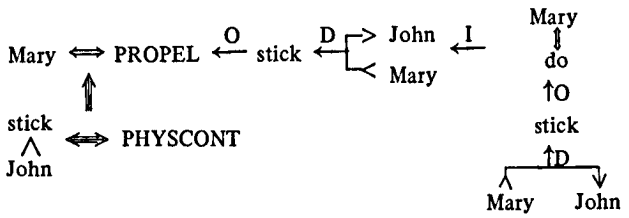
Since every ACT has an instrumental conceptualization that can be said to be part of that ACT, we can see that it should therefore be impossible to ever actually finish conceptually diagramming a given sentence. That is, every ACT has an instrument and so on. In this sentence we might have conceptually

something like: "John transed the book to Mary by moving the book towards Mary by moving his hand which contained the book towards Mary by grasping the book by moving his hand moving muscles by thinking about moving his muscles" and so on. Since an analysis of this kind is not particularly useful and is quite bothersome to write, we do not do so. Rather, whenever we represent a conceptualization we only diagram the main conceptualization and such instrumental conceptualizations as might be necessary to illustrate whatever point we are making. It is, however, quite possible that we might need many of these instrumental conceptualizations in a program that was intended to simulate certain body motions such as Winograd's (Winograd, 1971) block moving program. Thus, the ACT in a conceptualization is really the name of a set of actions that it subsumes (and are considered to be a part of it). These instrumental conceptualizations are not causally related since they are not actually separable from each other. In actuality, they express one event and thus are considered to be part of one conceptualization. The rule is then that one conceptualization (which may have many conceptualizations as a part of it) is considered to represent one event.

In ordinary English usage, the syntactic instrument of a given sentence usually corresponds conceptually to the object of a conceptualization that causes the conceptualization most directly related to the verb of which it is an instrument syntactically. Thus as an illustration of this we have:

Mary hit John with a stick.

We represent the conceptual action underlying 'hit' by PROPEL which means to apply a force to an object plus the resultant state PHYSCONT. Thus we have conceptually:



The 'do' in the instrumental conceptualization indicates that the action by which the PROPEL-ing was done is unknown. This corresponds to the fact that this sentence is actually ambiguous, the two most common interpretations being that 'she swung the stick' or that 'she threw the stick'. Representing such a sentence in this manner allows for the discovery of this ambiguity. (In an actual computer analysis schema the blank 'do's' can be realized as predictions about missing information which must be discovered either by inquiry or memory search.)

Predictions about what ACT's fit into this instrumental slot are made from the ACT in the main conceptualization. PROPEL requires either 'move' or

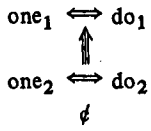
'move' + 'ungrasp' as actions for its first instrument. 'Swing' and 'throw' are mapped conceptually into 'move' and 'move' + 'ungrasp' respectively (with additional information as to manner).

It is interesting to consider how such a deep conceptual analysis of natural language utterances can help us in parsing and understanding those utterances. Consider:

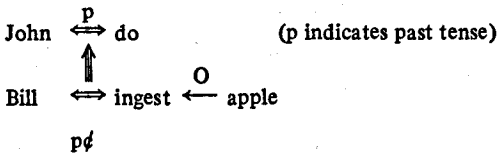
John prevented Bill from eating the apple.

The verb 'prevent' is conceptually a statement about the relationship of two events, namely that one event causes the inability of the occurrence of a second event. Unless we treat 'prevent' in this manner, important paraphrase recognition ability will be lost, and in addition even the ability to intelligently parse sentence derivative from this will be hindered.

Conceptually, then, 'prevent' is not something that anyone can do; rather, it expresses the following relationship between two events.



This is, person$_1$ doing something caused person$_2$ to not be able to ($\not{c}$) do something else. Thus we have:



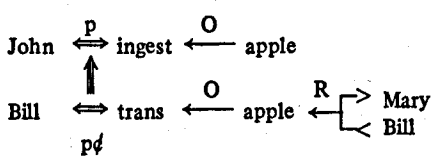
If we had an intelligent understanding system, we might want to know what John 'did' and this representation allows us to realize that we could ask that.

Now consider:

John prevented Bill's giving Mary the apple by eating it.

Along with the information that 'prevent' represents the conceptual structure shown above is a clue as to how to go about finding what might fill in the first 'do'. This clue is that if the ACT that replaces the 'do' is present it is most probably in the syntactic instrument of 'prevent', that is, in a by-clause.

Thus, that clue is used to give us:



It is important to notice that it is quite possible to realize the above structure in the following sentences as well.

> Bill couldn't give Mary the apple because John ate it.
> When John ate the apple it caused Bill to be unable to give it to Mary.
> When John ate the apple, it meant that Mary didn't get the apple.

The above sentences do not use 'prevent' in words but they do use the concept underlying 'prevent'. It is extremely important that any theory of understanding analyze these sentences or any of the myriad other paraphrases into only one conceptual structure in a natural way. This requires establishing the relationships between actual events rather than between the words that may have been used to describe those events. In order to do this, it is necessary to break words down into the primitive actions and events that they describe.

In summary, then, conceptual dependency is a representation for expressing the conceptual relationships that underlie liguistic expressions. The basic structure of this conceptual level is the conceptualization. A conceptualization consists of either an actor-action-object construction or an object-state construction. If an action is present then the cases of that action are always present. One case of an action is instrumental which is itself a conceptualization.

Conceptualizations may be related to other conceptualizations causally. Just as it is impossible to have an action without an actor so it is impossible to have the cause of a conceptualization be anything other than another conceptualization. (This means that 'John moved the table' must be conceptually 'John *did something* which caused the table to be in a different position'. This *doing* is not 'move' but rather something that was unstated. The doing can be inferred and is most probably 'apply a force to'.)

THE PRIMITIVE ACTIONS

Using the framework for language analysis that was just explained the total number of ACTs that are needed to account for any natural language sentence is twelve. In stating this, we are not claiming that this number is totally accurate. Rather, the claim is that the order of magnitude is correct and that these twelve ACTs or some set of ACTs not significantly different than those presented here are all that is necessary to represent the actions underlying natural language.

This result is caused partially by our rewriting a great many verbs into caused states conceptually. Nevertheless it is significant that so few ACTs are actually necessary to account for the basis of human activity.

There are four categories of ACTs that the twelve ACTs are broken down into: Instrumental (2), Physical (5), Mental (3), and Global (2).

Physical ACTs

The Physical ACTs are:

PROPEL
MOVE
INGEST
EXPEL
GRASP

It is our claim that these are the only ACTs that one can perform on a physical object. Furthermore, there are restrictions on what kinds of objects any given ACT will accept.

The meaning of the ACT and the objects are as follows:

PROPEL: means 'apply a force to'; its object must be under a certain size and weight, but for our purposes we will say that any object is acceptable.

MOVE: means 'move a bodypart'; the only objects that are MOVEd (in our sense of MOVE) are bodyparts.

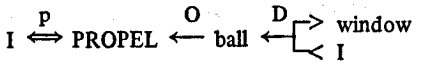
INGEST: means 'take something inside you'; INGEST's object must be smaller than the mouth of the actor or must be divided into pieces smaller than the mouth opening; object should be food.

EXPEL: means 'take something from inside you and force it out;' its object must have previously been INGESTed.

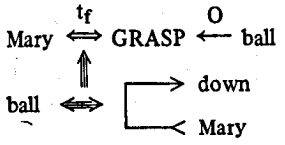
GRASP: means 'to grasp;' object must be within a size limit.

Some example sentences and their analyses are:

I threw the ball at the window.

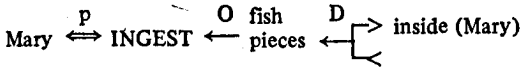


Mary dropped the ball.



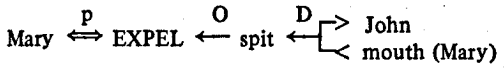
(where t_F means 'the end of' an action)

Mary ate fish.

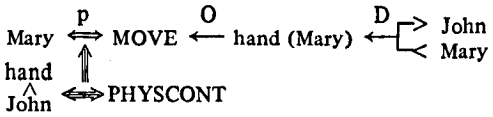


(parentheses denote the POSS relation)

Mary spit at John.



Mary touched John with her hand.

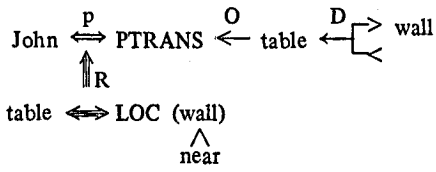


Global ACTs

As can be seen by the nature of the physical ACTs, very often an ACT is somehow more than the sum of its parts. That is, often the result of an ACT is focused on more directly than the ACT itself. Since the representations presented here are intended to represent human thought it is necessary to do the same focussing that humans do. We thus use the notion of Global ACTs which express the change of state consequences and intentions of a variable physical ACT.

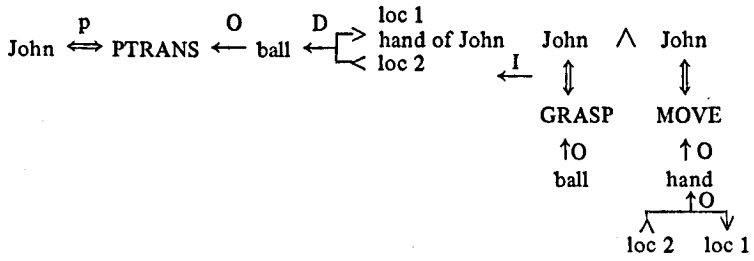
The most important Global ACT is PTRANS. PTRANS expresses the change in physical location of an object. In order to change the physical location of an object it is necessary to perform one of the physical ACTs upon that object first. That is we can have:

John moved the table to the wall.



and

John picked up the ball.



(loc 1 higher than loc 2)

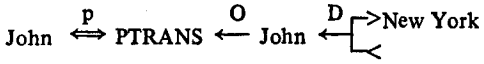
Since PTRANS is of such importance in Conceptual Dependency analysis it is worthwhile to spend some time discussing it. While the use of PTRANS for change of location verbs such as move and pick up is farily straightforward, we also use PTRANS to represent the ACT underlying the verb 'go'. This is a difficult point for speakers of English to accept and thus requires some explanation.

Most semantic analyses deal with 'John went', 'the car went', and 'the plane flew' as if the sentential subject is also the actor or agent semantically. In fact 'John' is the actor in 'John went'. What is important to realize is that 'John' serves a dual role conceptually here. 'John' is also the object of the sentence 'John went'. In saying this we pay careful attention to the problem of inference from a conceptual analysis.

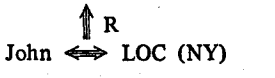
Since the conceptual representations that we are proposing here are used by a computer that is attempting to understand, it is important that the representations be consistent so the programs that operate on them can be general. One generality that we use is that whenever PTRANS is present, it can be inferred that the object of PTRANS is now located at the location present as the directive case for PTRANS.

Thus since it is true that John is the actor when he 'goes', 'John' must be in the actor slot. But, it is additionally the case that the location of John has been changed and that, just as for 'move' and 'pickup', John is now probably located at the directive case location.

Thus the sentence: John went to New York. is conceptually analyzed as:



Actually, this indicates that the direction is towards N.Y. The completed act requires a generated state result. Here we would have:

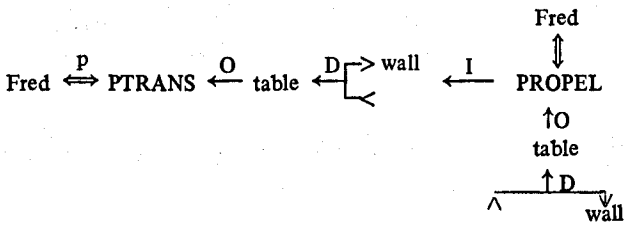


(That is, John is in New York.)

It can be seen that whenever PROPEL is present PTRANS can be inferred. Thus for:

Fred pushed the table to the wall.

we have:



The most abstract of the global ACTs is ATRANS. The objects that ATRANS operates upon are abstract relationships and the physical instruments of ATRANS are rarely specified. The 'trans' that was referred to in the beginning of this paper is what we call ATRANS. ATRANS takes as object the abstract relationship that holds between two real world objects. We have:

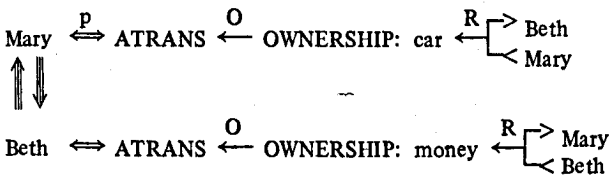
Mary gave the book to John.

Mary loaned the book to John.

In other words, ATRANS changes one of the parts of a two party abstract relationship. ATRANS can be actually effected in the real world by many means, not all of them physical. The most common instrument for ATRANS is 'MOVE $\overset{O}{\leftarrow}$ hand' where the hand is grasping the object being transferred. Often, however, OWNERSHIP is transferred by signing a paper or by simply saying so. That is, ATRANS can take place and the world can appear exactly as it was to an untrained observer. For this reason, ATRANS is the one ACT presented here that is not necessarily universal. That is, it is possible to conceive of a culture and therefore a language that has no notion of possession and therefore has no ATRANS.

ATRANS operates with a small set of abstract objects. We treat 'sell' as a change in the ownership relations:

Mary sold her car to Beth.

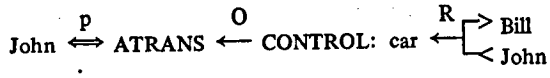


Thus, we are saying that two abstract relationships changed because of some mutual causality. Any physical ACTs that took place (i.e., signing a check and handing it to Mary) are the instruments of the abstract action ATRANS.

We use the verb 'give' in English to denote the change of these abstract relationships. 'John gave the ball to Bill' is a change of possession so ATRANS is used:

Another abstract relationship that can be ATRANSed is 'control'. Thus when we say 'John gave his car to Bill', the most likely interpretation is that this is an ATRANS of control rather than ownership. 'ATRANS $\overset{O}{\leftarrow}$ CONTROL' then, is to 'give the use of.'

John gave his car to Bill.



The problem here is that the use of the above primitives makes clear an ambiguity that exists in English that is not otherwise always accounted for in semantic representations. Namely, 'give' can mean a change in possession that required no physical change as in 'Mary gave John the Empire State Building'. 'Give' can also refer to a change in control without a change in possession. Additionally, 'give' can refer to a change in physical location without a change in the abstract notion of possession, as in 'I gave him my handkerchief'. Basically, then, whether 'give' means ATRANS or PTRANS or both is dependent on the nature of the object and is often simply ambiguous. A great deal of the information needed to process languages is based on the thing involved rather than the action.

Instrumental ACTs

There are two instrumental ACTs:

ATTEND
SPEAK

These ACTs are not very interesting in that they are used almost totally as the instruments of some other ACT.

SPEAK is the ACT which actually produces sounds and its objects therefore are always 'sounds'.

ATTEND takes sense organs as objects and physical locations as directions. So 'ATTEND eye' is 'see', 'ATTEND ear' is 'hear', and so on.

Mental ACTs

The three mental ACTs are:

MTRANS
MBUILD
PLAN

Since these ACTs are by no means straighforward, we shall spend some time discussing them.

MTRANS

The ACT MTRANS is meant to handle the flow of information to and from the conscious mind. It, plus various mental building acts, should serve to represent all the ways in which we bring thoughts into our heads.

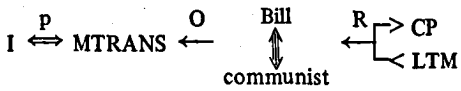
MTRANS represents a change in the mental control of a conceptualization (or conceptualizations) and underlies verbs like recall, commit to memory, per-

ceive, sense, and communicate. It has several features different from the physical PTRANS. For one, the object that is MTRANSed does not leave control of the donor, but is copied into the control of the recipient. Further, the donor and recipient are not always two different people but often two different mental processors (or locations: the distinction in the mind is as fuzzy as the distinction between program and data in the computer), which are frequently within the same person. Five such processors are used:

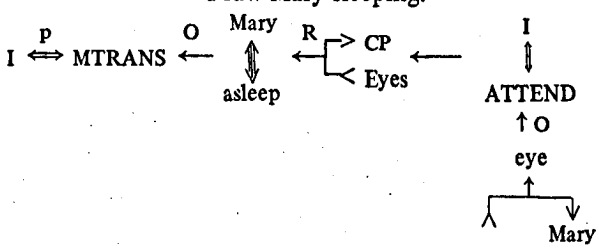
1. Conscious Processor (CP) – is the storage place for whatever is currently being processed.
2. Long Term Memory (LTM – this is primarily the store of beliefs one has about the world. It is a processor too, where such actions as forgetting and subconscious association occur, but the level of activity is both low and hard to characterize, so it shall be treated as a passive element here.
3. Immediate Memory (IM) – this is a storage area for the current context being processed as well as the processing area for conscious thought.
4. Sense-Organs (Eye, Ear, Nose, Tongue, and Skin) – these are all preprocessors, converting raw sense data into conceptualizations describing that data.
5. Body – this covers whatever processors handle internal sensations such as pain, unease, excitement, etc.

With these items, we can handle many mental verbs, such as

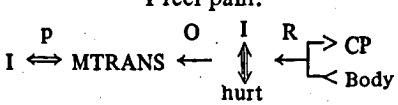
I remembered Bill was a communist.



I saw Mary sleeping.

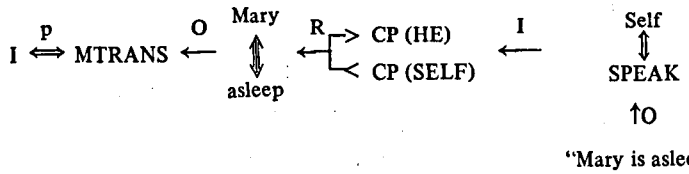


I feel pain.

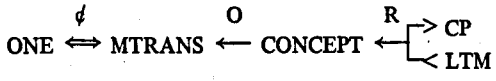


This use of MTRANS covers mental actions where the concept brought into awareness has been internally arrived at, rather than externally generated. Some external MTRANSes are:

I told him Mary was asleep.

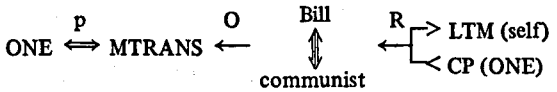


Forgetting is simply the inability to bring something from LTM:



Verbs such as 'learn' and 'teach' also involve MTRANS to LTM from CP. Thus:

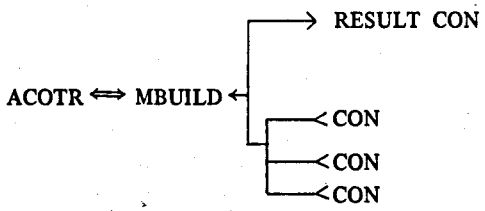
I was taught that Bill was a communist.



That is, 'teach' is really like communicate. The actual difference lies in the fact that the communicated information is said to be new in the case of 'teach'. Thus we also have the information that this information was not in the LTM of I before.

MBUILD

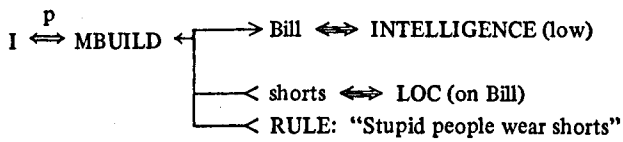
The ACT MBUILD accounts for thought combination. MBUILD is written as:



MBUILD takes as object a many-to-one 'functional' arrow that denotes the

combination and transformation of several units into one resultant unit. MBUILD plays the role of the action which is antecedent to some more "final" act of accepting the result as knowledge or as a belief. Examples of this type are 'conclude', 'resolve', 'prove to oneself', 'solve', and so on. In others of these, MBUILD is the only ACT underlying the verb, and there is no result conceptualization yet produced (such as 'think over', 'consider', 'reason out', 'relate', etc.). This distinction between the process and the result of the process (and what becomes of the result afterward) is crucial to the unravelling of mental verbs. MBUILD refers only to the process of combination, or *attempted* combination, and includes no information about the success or failure of the operation. Success can be denoted by the presence of a result in the object slot, and failure by its absence.

I decided that Bill is stupid because he wears shorts.

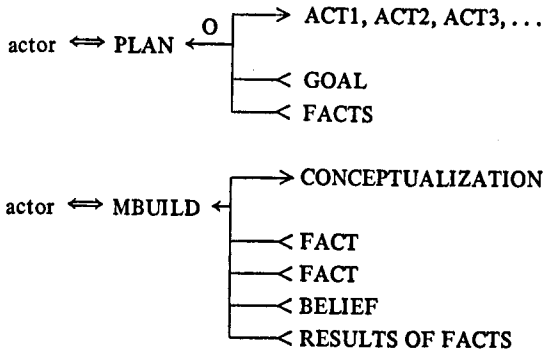


PLAN

PLAN is intended to account for an individual's ability to decide on a step-by-step course of action that leads to a goal. PLAN takes as input a goal and a set of possible plans that are known to be ways to achieve that goal. PLAN decides among them and produces as output a sequential chain of conceptualizations that the actor intends to act out in order to achieve the desired goal. PLAN takes MBUILD as its instrument where the items MBUILDed are the particular decisions that make up the selection of a given plan.

The general form of PLAN is with an object arrow that takes as input a goal and the plans associated with that goal and produces as output the chain of acts that the actor intends to perform to achieve the goal.

The general form of MBUILD is with an object arrow that takes multiple inputs dealing with the current facts and the logical consequences and beliefs associated with those facts and produces a new thought which may or may not be an intended future action.



The difference in these two ACTs with respect to their realization in English is as follows:

John decided to go to the Bahamas.	MBUILD new thought
John developed a plan to go to the Bahamas.	PLAN unknown set of ACTs. Goal that is input to ACT PLAN is known.
John knows how to get the money.	PLAN with known goal.
John figured out who the murderer is.	MBUILD new thought
John did that to get Mary to like him.	PLAN
John intended to hurt Mary.	PLAN (ambiguous if no ACT was done)
John intends to hurt Mary.	Ambiguous (Not known if he has figured out how)

INFERENCES

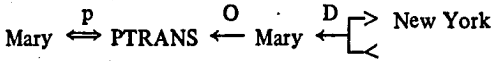
It should be clear that any attempt of this kind to put sentences into underlying representations that use only a few primitive ACTs must have as its intent the use of these ACTs in some prescribed fashion. Each ACT is basically a memory affector, in that whenever that ACT is present certain facts can be inferred from it.

In considering the problem of how to know when something would qualify as a new ACT, the pertinent question to ask is whether the inferences that would be drawn from that ACT are the same as the set of inferences that are drawn from some already existing ACT.

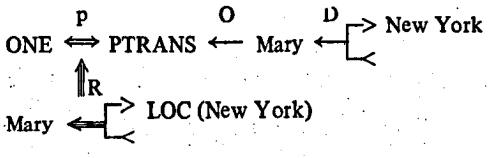
Here it is important to make clear what exactly we mean by an inference. For our purposes, an inference is a conceptualization that is true to some degree of probability whenever some other conceptualization or set of conceptualizations are true. For example, in the sentence

Mary went to New York.

it is not explicitly stated that Mary in fact arrived in New York. The sentence is graphed as:



while 'Mary arrived in New York' is graphed as:



that is, we don't know if she actually got to New York. We know only that she

went in that direction. We infer that if we are told something and not explicitly told that the expected inference is invalid, then it is reasonable to draw that inference. In this case PTRANS causes the location inference to be generated in absence of information to the contrary.

Essentially, all the Conceptual Dependency is oriented to how the representations will best be used. They will be used in memory for inferencing. We conceive of the language understanding process as a set of procedures that seek to extract what is explicit in an input and attach it to the body of knowledge associated with that explicit information in a way that will allow the system to formulate questions that will fill in the "holes" that exist in the final meaning representation of that input. People understand more from a sentence than they have been told explicitly. They also understand what they have not been told and what they must infer in order to make sense of what they have been told. Knowing what you don't know allows for the processes to be generated that will fill in the empty holes.

What we are saying is that human communication is based largely on what is left out of a discourse. People rarely specify everything they intend to communicate. Rather, they specify enough to lead the hearer to an understanding of what was meant. They leave out anything that they assume the hearer can figure out for himself. It is easy to see that the problem for an understander is the recreation of what has been left out.

We call the propositions that have been left out *inferences*. Simply defined, an inference is anything that is likely to be true about a given input but is not necessarily true. A rule of thumb that can be used in determining what is and is not an inference, we call the *but test*. A proposition is a reasonable inference from another proposition (in English) if the negation of that proposition preceded by the word 'but' in conjunction with the first sentence yields a sensible sentence. If the result of this 'butting' yields a sentence that denies its own premise, then the second proposition is said to be *implicit* within the first. If the result of the conjunction is nonsense because there is no obvious relation, then the second proposition is not an inference from the first. Some examples of this are:

John punched Mary but his hand didn't touch her.	*implicit*
John punched Mary but she didn't cry.	*valid inference*
John punched Mary but it didn't rain.	*invalid inference*

The question of understanding sentences like the last one here is not to be dismissed by marking it invalid. If somebody said the last sentence we would want to know why he expected that such a consequence would result. (For the purposes of the test we simply show that raining is not usually to be inferred from punching.)

Inferences, then, are never certain. They come from rules that a hearer learns to apply to various words, concepts, and situations. Inference is the core of the understanding process and thus inferences lie at the center of human communi-

cation. They serve to tie together inputs into a related whole. Often the inferences themselves are the main point of the message.

If inferences are such a crucial part of understanding, we might wonder why they are left out of spoken language. The answer is simply one of space. People try to be as concise as possible and avoid putting in remarks that are obvious to the hearer. This process involves assessing what the hearer knows while talking to him. This assessment is not always made well, and if what has been left out is important and must be inferred, a misunderstanding can result. Inferences are also left out because the speaker doesn't want to have been said to have said them. Politicians make use of this inferential capability of hearers all the time.

The process of inference is wholly dependent on an adequate supply of world knowledge. Without the knowledge about what can happen in the world, it is impossible to make sense out of what has happened. Small children are capable of witnessing earth-shaking events without making sense of them because of their lack of knowledge about the implications of these events. That is, they didn't know what they were seeing.

Rieger (Rieger, 1975) has classified the process of inference into sixteen distinct inference classes. It is his thesis that every input sentence is subjected to the mechanisms that are linked to these classes to produce inferences every time a sentence is received. Below are Rieger's classes of inferences:

1. Specification – What parts of the meaning underlying a sentence were implicit and must be filled in?
2. Causative – What caused the action or state in the sentence to come about?
3. Resultative – What are the likely results of an input action or state in terms of its effect on the world?
4. Motivational – Why did the actor perform the action? What did he intend to happen?
5. Enablement – What states of the world must have been true for the actor to perform his action?
6. Function – What is the value or use of a given object?
7. Enablement/prediction – If a person wants a state of the world to exist, what action will it then be possible to perform?
8. Missing enablement – If a person can't do what he wants, what state will have to change in order to permit it?
9. Intervention – What can an actor do to prevent an undesirable state from occurring?
10. Action/prediction – Knowing a person's needs and desires, what actions is he likely to perform?
11. Knowledge propagation – Knowing that a person knows certain things, what else is he likely to know?
12. Normative – What things that are normal in the world should be assumed in the absence of being told them specifically?
13. State-duration – How long will a given state or action last?

14. Feature – What can be predicted about an entity when a set of facts is known about it?
15. Situation – What other information can be assumed about a given situation?
16. Utterance-intent – Why did the speaker say what he said?

It is memory's responsibility to establish the inferences that are likely to be true in a given situation. These inferences are made partially because it is often the inferences themselves that are the point of the input. Inferences are also made because they will lead to questions and knowledge that will lead to the understanding of a sequence of sentences as a whole.

The basic mechanism in understanding is the inference process. People understand more than what they have explicitly been told. They do this by tying together what they have heard with what they know about what they have heard. We can illustrate this interaction of knowledge and inference in understanding by considering the following sentence:

John saw a large boulder rolling down the mountain towards his house.

followed by each of these sentences as the second sentence in a two-sentence paragraph:

1. He ran inside to get his pet mouse.
2. He started digging a path that led away from his house.
3. He hid under his bed.
4. He picked up the telephone.
5. He started to cry.
6. He ran inside to get a mattress.
7. He yelled for his wife.
8. He called the geological society.
9. He started to write his will.

The process of understanding is considerably more than just deciding upon the appropriate meaning for a given individual sentence. The meaning of a paragraph is more than the sum of the meaning of each individual sentence contained in it. People leave out from a story or sequence of sentences the reasons for each action in the sequence and the way in which the individual items in the sequence relate to each other. These things are left out largely because they are obvious. We could ask a person who has just heard any of the nine sequences above why the action communicated in the second sentence was done and they would probably give answers of the following sort:

1r. Because he wanted to save the mouse's life.
2r. To cause the boulder to veer away from the house.
3r. He was frightened and irrational.

4r. He intended to call for help.
5r. He was frightened and gave up hope of saving himself.
6r. Maybe he though he could cushion the blow of the boulder by putting the mattress in front of the impact point.
7r. He wanted to make his wife aware of the problem so she could leave the house and save herself.
8r. Perhaps he thought the rock was an interesting specimen and he wasn't aware of the danger.
9r. He assumed he was going to die and wanted his possessions dispersed according to his wishes.

These responses indicate an understanding of the situation being discussed apart from and in addition to what has actually been related. Within the process of understanding, memory is responsible for finding the knowledge that it has that is relevant to what it has heard and using that knowledge to make inferences about the intentions, motivations, and effects of the actions in the sequence that it has heard.

What is obvious to a person is of course not obvious to a computer. The problem of getting a computer to understand is at least partially the problem of finding out what information is necessary when. In order to understand the sequence above using sentence (1) as the second sentence, it is necessary to access the facts that:

a. Large objects going at a sufficient rate of speed can destroy objects of insufficient rigidity.
b. A boulder rolling down a mountain is an instance of the first part of fact (a) and a house is an instance of the second part.
c. Things inside objects about to be destroyed often get destroyed as well.
d. People try to prevent actions that have bad consequences for them.
e. People value their possessions and dislike losing them.
f. Taking something away from a situation that will destroy it will prevent that destruction.
g. Pets are valued possessions.

A combination of these seven facts will help to process this two-line sequence. Only facts (a) and (b) are used in every sequence. The other eight sequences require facts of their own to make sense out of them. Thus, it is obvious that part of the problem of understanding is organizing the knowledge that one has so that it is possible to find what is needed in a given situation. A model of the understanding process must necessarily be extremely complicated. The base of any solution must contain a memory for facts that will apply as inferences in a given situation.

Once natural language sentences can be reduced to the conceptualizations

underlying them with the use of primitive actions, the inference process is simplified. We are guaranteed to have activated all parts of the semantic equivalence class if any of its members is activated. The problem of inference is by no means completed by the use of these primitives. What we have done is to reduce the number of inferences that need be stored by rewriting, so to speak, the verb into an ACT from which we can draw inferences. Certain inferences are simply not taken care of by this. For example, if we have 'Mary kissed John', our mapping of kiss into 'MOVE lips towards' will not simplify the problem one bit (most inferences fall into this category, in fact). One must be careful not to lose information in doing a conceptual analysis. (That is, 'kiss' is really more than just 'MOVE lips towards'.) However, the mapping of the various verbs into ATRANS, for example, eliminates the problem of having to make the same inference over and over again.

The value of these primitive ACTs is that certain things are true whenever a given ACT is present and thus large amounts of information that is true for a given verb can be written only once for that underlying ACT. These equivalence classes, then, are probably much more like what people learn than would be an exhaustive list of what is true for every verb.

In addition, verb paraphrasing is explained by the use of these primitives. The core of the MARGIE program for paraphrase and inference was, of course, the notion of primitive ACTs.

MARGIE

The MARGIE program was written at Stanford to demonstrate the parser, inference and generation programs that were written based on Conceptual Dependency Theory. The MARGIE (Memory, Analysis, Response Generation, and Inference on English) computer program was designed and built at the Stanford Artificial Intelligence Project as a first attempt at dealing with the problems of handling meaning and inference. The program had three separate pieces. The conceptual analyzer (Riesbeck, 1975) took a subset of English strings and mapped them into a deep conceptual representation of their meaning. The memory and inference program (Rieger, 1975) stored new inputs, established references, and made inferences about the input it had just received. The third piece was the generator (Goldman, 1975) which encoded conceptual representation into syntactic structures. A modified version of a program written by Simmons and Slocum (Simmons and Slocum, 1972) was then used to map the syntactic structures into English sentences.

MARGIE had two modes, PARAPHRASE and INFERENCE. In PARAPHRASE mode, sentences were analyzed into a deep representation of their meaning and then that meaning was "read out" in as many ways as were available to the generator for expressing that meaning in English. Since the deep representation used constructs that were not part of English, the problem was one of coding ideas into language, rather than that of transforming input

sentences in a variety of ways. In fact, the actual input sentence was not retained in the paraphrase program after its meaning had been ascertained.

In INFERENCE mode, input sentences were analyzed into their meaning representation and then operated on to find out what other information might have been true or arisen as a consequence of the input. Inferences were established and used as the basis for forming still more inferences. These inferences were then encoded into English and output.

MARGIE took up about 75K of core in its compiled version. It ran on the PDP 6/10 at Stanford. The response time was a few seconds in PARAPHRASE mode and a few minutes in INFERENCE mode.

The significance of MARGIE lay in its possibilities for expansion rather than any present-day technological advance. It was limited to somewhat stilted sentences and its vocabulary and world knowledge were very small. We have presented this work because the basic components of MARGIE are likely to be the basic components of any future computer understanding system, and it is in this sense that this work is best understood. Paraphrasing and inferencing as we had defined them in MARGIE are non-tasks. That is, they are not things that people do, nor that are important for machines to do. However, the individual pieces of MARGIE were things that are important to model. That is, meaning analysis, inferencing, and generation are important sub-tasks on the road to computer understanding.

Examples

It is helpful, in order to ascertain the scope of the program, to look at some examples of what it could do. The examples presented here were all run on the Stanford computer. The sentences presented to MARGIE were in no sense random in that the user was of course aware of the vocabulary of the program.

INFERENCE MODE

INPUT: John told Mary that Bill wants a book.
OUTPUT1: A book about what?
OUTPUT2: Mary knows that Bill wants a book.
OUTPUT3: Bill wants to come to have a book.
OUTPUT4: Bill wants someone to cease to have a book.
OUTPUT5: Bill wants to read a book.

INPUT: John gave Mary a beating with a stick.
OUTPUT1: A stick touched Mary.
OUTPUT2: Mary became hurt.
OUTPUT3: John wanted Mary to become hurt.
OUTPUT4: What did Mary do to make John angry?

INPUT: John gave Mary an aspirin.
OUTPUT1: John believes that Mary wants an aspirin.

OUTPUT2: Mary is sick.
OUTPUT3: Mary wants to feel better.
OUTPUT4: Mary will ingest the aspirin.

INPUT: John is going to go to the store.
OUTPUT1: John wants to be at the store.
OUTPUT2: John wants to buy something.
OUTPUT3: John will trade money to the store for something.

PARAPHRASE MODE

INPUT: John advised Mary to drink the wine.
OUTPUT: John told Mary that drinking the wine would benefit her.

INPUT: John killed Mary by choking Mary.
OUTPUT1: John strangled Mary.
OUTPUT2: John choked Mary and she died because she could not breathe.
OUTPUT3: Mary died because she was unable to inhale some air and she was unable to inhale some air because John grabbed her neck.

INPUT: Reading the book reminded Rita to loan the book to Bill.
OUTPUT: Rita remembered to give the book to Bill and she expects him to return it to her because someone read the book.

INPUT: John prevented Bill from giving a banana to Mary by selling a banana to Rita.
OUTPUT1: Bill was unable to give a banana to Mary because Rita traded John some money for a banana.
OUTPUT2: Because Rita bought a banana from John Mary could not get a banana from Bill.

CAUSALITY

After MARGIE was finished, we began to work on the problems of the representation of text. It was quite unreasonable to be working on individual sentences as an end in themselves. The first problem we encountered was that of causality.

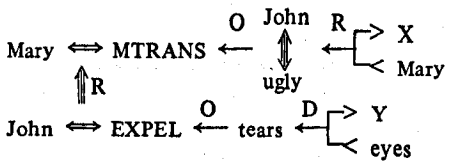
Causality problems are, of course, present in individual sentences as well, so as a first step in processing text, we went right back to dealing with individual sentences. The real problem is connecting up of ideas and events.

Consider the following sentence:

John cried because Mary said he was ugly.

It is possible to consider this sentence as having underlying it the conceptualizations 'John cried' and 'Mary said John was ugly'. We have treated this in Con-

ceptual Dependency as an example of what we had called 'reason causation' (denoted by ⇑R). Thus, we would have treated the above sentence as:



A very simple and obvous thing that is incomplete about this representation is that we must assume that John heard this statement of Mary's, either from Mary or from someone else, yet this is not explicitly stated.

So the first inference that must be made is that 'John is ugly' was MTRANSed to John. What else do we know about this sentence? We know that something must have gone on in Johns's mind concerning Mary's statement. We can guess that the statement must have caused him to think about the fact that he was ugly and thus made him unhappy or perhaps that Mary didn't like him and he wanted her to, and this made him unhappy. Thus, though the above analysis is superficially acceptable, conceptually it must have been something like:

'Mary MTRANSed, caused John know what
Mary MTRANSed, caused John think about some set
of facts, caused John be unhappy, caused John EXPEL.'

Two problems come to light considering this new analysis. One is that these causals are quite different from each other. The fact that the English word 'cause' can be used here helps to cover up the problem. As will be shown later, these conceptualizations affect each other quite differently. It is necessary, then, to delimit the types of causality that can be used conceptually.

The next problem is more philosophical. Do we really need to do such an extensive analysis? What does it actually buy us? We claim that it is almost always necessary in a complex understanding task to go as deeply as possible in order to provide some universal base that is language-free with which to work. Clearly, for certain tasks, the above sentence need not be broken down any further than we have done here. However, consider the following sentence (discussed in detail with respect to a computer program that handles it in (Rieger, 1975)):

Mary kissed John because he hit Bill.

A program that failed to notice that Mary was probably pleased by Bill's getting hit would have failed to understand what is one of the most interesting facts in that sentence. In order to be able to make such an inference, it is necessary to break up the surface causal relation into one that is more reflective of the causal connections of ideas. With this introduction we shall now proceed to the syntax of causality.

Causal types

We allow four kinds of causality each occurring in the following conceptual syntactic arrangement:

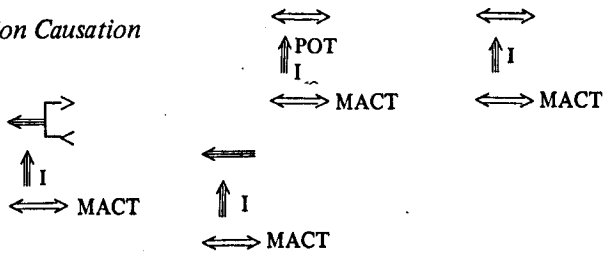
Result Causation

A conceptualization involving an ACT (denoted ⟺) can have a *result* (⇑r) that is a change in value of some state of an object.

Enable Causation

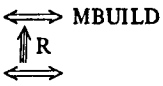
When a state change occurs, it can possibly complete sufficient conditions for an ACT to take place. This is called *enable causation* (⇑&). Such state changes only enable potential ACTs ($\Longleftrightarrow_{POT}$). A potential ACT is an abstraction that can be realized as an ACT if the actor so desires.

Initiation Causation



Whenever an ACT or a statechange occurs, or whenever a state of a POT-ACT exists, it is possible that an actor may be made aware of it, and made to think about it. This is called *initiation causation* (⇑I). Initiation causation accounts for people thinking about things.

Reason Causation



Once people have started thinking about things, they are likely to decide to do something (MBUILD). Deciding to do something is the *reason* for doing it. This is *reason causation* (⇑R).

Lead-to causation: One event can be said to *lead-to* another event. This is solely an abbreviation for one of the above causals. It is not always important to know how to expand a causal so we must provide some means of representing this conceptually: (⇑L).

Causals causing: Certain causals can themselves cause or be caused.

Negatives causing: A negated conceptualization cannot really cause anything. However, a non-event can initiate a thought, as can anything else, so we allow negatives to cause MBUILDs in Initiation Causation.

A classification of causal relations according to conceptual syntactic type is desirable for both psychological and computational reasons. Certainly it is true that since the different causal arrows postulated related different combinations of conceptualizations, then we would logically consider that there is only one causal arrow. It has been our philosophy to try to represent differences that seem psychologically plausible without regard to their most logical representation. We feel that the ease of psychological (as opposed to logical) inferencing which such a method provides is important.

The causal types above can be considered to be different relational arrows between conceptualizations. We could treat these essentially syntactic distinctions as conceptual semantic distinctions. That is, we could allow only one causal arrow, with complicated rules relating what must be on one side of the arrow to what was on the other side. From the point of view of prediction in analysis such an approach would be computationally less efficient. Certain words in English (and other languages) refer to only certain of the causal arrows and not others ('allow' and 'prevent' refer to "enable causation," for example). Knowing the conceptual syntactic type of a causal (from the input word) thus delimits the possible semantics. Information predicted on the basis of the presence of one causal type about what other causals would be present allows for a better top down search in the analysis. Thus, if we know that two conceptualizations have been related by an English 'cause' word, then the syntax of causality predicts what type of conceptualization must intervene between them in a causal chain. This enables us to search for them either by looking in the sentence or by asking questions.

Such a causal classification makes inference simpler as well. Given X causes Y, where the causal type has been determined, certain inferences are valid for one causal and not for others. For example, if enable causation has been determined on conceptual syntactic ground, it can be inferred that the other conditions that contributed to the final sufficient condition in the causal were also present. This can be done directly from the syntactic type without having to first infer that the most recent state change completed the sufficiency conditions.

Other causal types

The above causal relations can have certain changes made to them that cause them to behave differently. Any causal can be stated as being potential. This is

symbolized by a 'c' on the causing conceptualization. A (c)⇐B is considered to be 'if A then B would occur'.

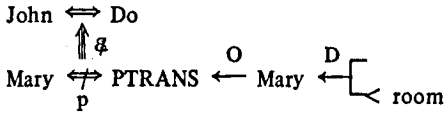
In addition to this, enable causation can be negated (symbolized $\not c$). This means that a state change can remove the sufficient conditions for an ACT to take place. This, in effect, is another kind of causation (unenable causation); but since all the conditions and properties of enable causation are the same as unenable causation, we will treat them as being the same here.

It is not possible to place time modifications on causals themselves because causals are merely relations between conceptualizations. The conceptualizations denote events which themselves have times. Thus event 1 can be causally related to event 2 even if event 2 is a future event, and event 1 is a past event. In that case, the causality is hypothetical (will cause).

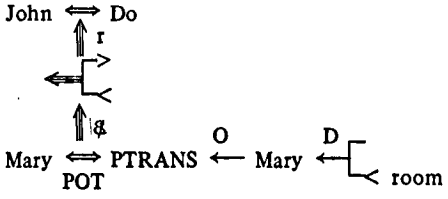
Use

Using this syntax, then, certain causal relations can be shown to need further explication. Thus, above we treated 'prevent' as:

John prevented Mary from leaving the room.



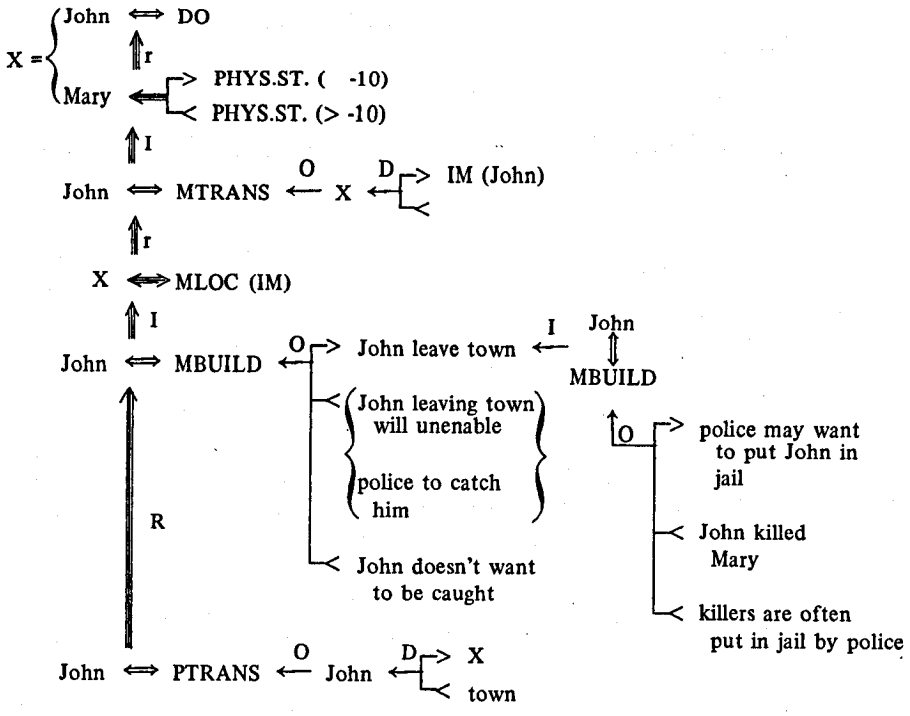
Now it can be seen that since conceptualizations of action (⟹) cannot cause other (⟹'s) (with the exception of Mental ACTs), this conceptual diagram must be further expanded. The actual diagram should be:



That is, it is the result of the ACT that John did that served to end the sufficient conditions for Mary's leaving the room. (For example, he might have broken her legs. Her non-movable legs would have ended the sufficient conditions for her leaving the room.)

John left town because he killed Mary.

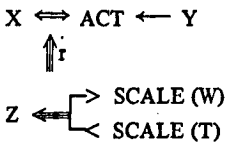
would be diagrammed as:


X = John ⟺ DO
r
Mary ⟸ PHYS.ST. (-10)
PHYS.ST. (> -10)
I
John ⟺ MTRANS ⟵ O X ⟵ D IM (John)
r
X ⟺ MLOC (IM)
I
John ⟺ MBUILD ⟵ O John leave town
John leaving town will unenable
police to catch him
John doesn't want to be caught
I John
MBUILD
O police may want to put John in jail
John killed Mary
killers are often put in jail by police
R
John ⟺ PTRANS ⟵ O John ⟵ D X
town

This example may seem fanciful, but we would claim that something like it is understood by hearers of the sentence and the mechanisms for creating such a conceptual structure are an integral part of an understanding system.

The most important part of the delimitation of how causals can connect rests with the semantics of causation. Thus, it is necessary to list every ACT and state combination that is permissible. By doing this, we are invoking a world model that in no sense can be said to be universal. Different cultures perceive different reasons for why things happen. As much as possible, then, these semantics are intended to indicate Western culture at best or my own personal culture at least.

Result Causation



a) The state change occurs in the object only (Y = Z) with the following exceptions:
   1) If the ACT is MOVE or GRASP the actor (X) can be Z.
   2) If the ACT is INGEST the actor (X) can be Z.

b) The following ACTs are the only ACTs that can cause a state change. The SCALES that can be changed are listed for each ACT.

PTRANS	LOCATION	(LOC)
	CONTAINMENT	(CONT)
ATRANS	POSSESSION	(POSS)
	OWNERSHIP	(OWN)
	CONTROL	(CTL)
MTRANS	MENTAL LOC	(MLOC)
PROPEL	PHYSICAL STATE	(PHST)
INGEST	PHST, HEALTH	
GRASP	PHST	
MBUILD	MLOC	

Examples where only result causation is present are:

John hit Mary $\xrightarrow{\text{result}}$ Mary is hurt.
Mary went to New York $\longrightarrow$ Mary is in New York.
John dropped the glass $\longrightarrow$ The glass is broken.

Mary kissed John $\longrightarrow$ Mary is happy.

Enable Causation

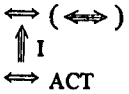
The state that enables an action must, of course, relate to that action. Thus, only the following states may enable the ACTs listed with them.

LOC	PTRANS, PROPEL, GRASP, MOVE, EXPEL, INGEST
POSS	ATRANS
CONTROL	ATRANS
OWN	ATRANS
PHYS. ST.	any ACT
HEALTH	any ACT
MLOC	MTRANS, CONC, MBUILD
ANGER	PROPEL

Enable causation is involved in the following examples:

He picked up the ball and then he threw it.
Mary told John what she had read in her book.
John found a ring and he gave it to Mary.

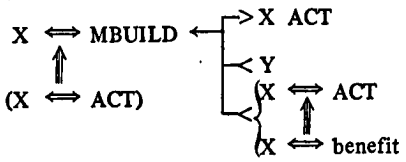
Initiation Casuation



Since it is possible for any event or state to cause someone to think of just about anything, there really exists no semantics for initiation causation, except the fact that the only ACTs that can be initiated are MTRANS and MBUILD. It should be possible though to trace a path from the initiating event or state to the new thought. Examples:

John reminded me of Bill.
A falling apple was responsible for Newton's Gravitation Law.

Reason Causation



The semantics of initiation causation are severely restricted since only MBUILD can serve as the ACT in the causing event. The only other restriction is that the causal event must have been present in the output object of MBUILD and that some set of conceptualizations (Y) must have been used in the MBUILDing.

Included within Y must be either a REASON or a BELIEF. A REASON is a conceptualization of the form 'One should do Z if W'. A reason must be present if an ACT is caused.

If a state change is caused, a BELIEF must be present within Y. A BELIEF is of the form 'Z is bad' or 'What is bad for W is bad for X'. Examples:

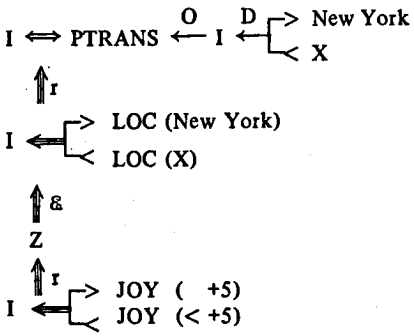
Mary hit John because she dislikes him.
John was unhappy because Mary died.

What all this means

Suppose we had the following sentences:

I am happy because I went to New York.
John's cold improved because I gave him an apple.
John died because Mary gave him a book.

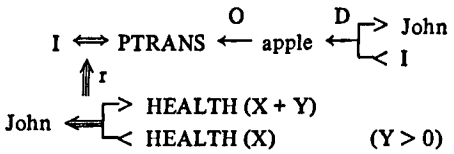
The conceptual representations of these sentences will now be more than the simple connecting of the stated events. For the first we have:



That is, PTRANS cannot change results having to do with JOY. Rather, only LOC can be affected. It must be inferred therefore, that the new location enabled the other ACT to take place which could affect the JOY scale. The only ACT which can affect the JOY scale (JOY is a MENT-ST) is MBUILD. Thus, we can infer that some other ACT initiated the MBUILD.

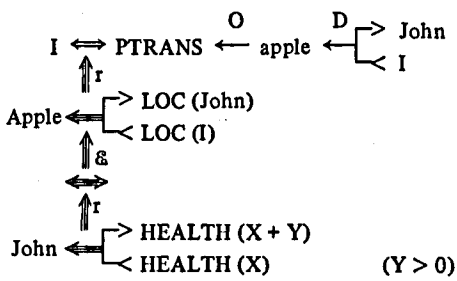
In the 'apple' sentence we are faced with a sentence that again 'really' means something different from its surface meaning. In order to interpret it we shall make use of the distinction between conceptual syntax and conceptual semantics.

It is important to realize that what has been presented so far is really idiosyncratic. That is not to say that many speakers may not share the particular world view adopted here, but it is clear that it is not necessary to do so in order to speak. The conceptual semantics is the idiosyncratic part rather than the conceptual syntax. Thus, the syntax says that PTRANS is an ACT and ACTs can have results. Thus, according to the conceptual syntax only, it would be possible to diagram this sentence as:



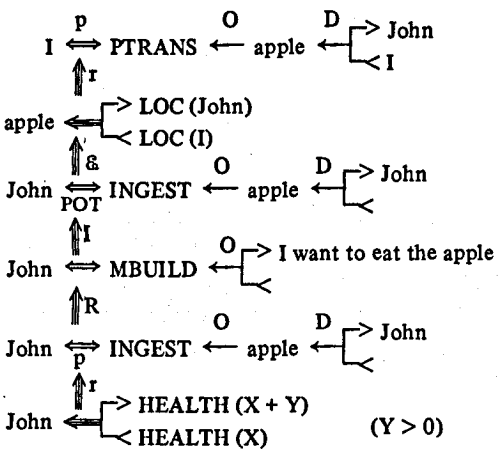
But, in fact, our conceptual semantics of causality explicitly disallows such a result causation. PTRANS can only affect LOCATION and CONTAINMENT in our particular world. It is certainly possible that there exist worlds where the mere possession or nearness of something can affect health, but in order to understand a person it is usual practice to impose our own world view on what he says. Thus, our analysis of this sentence does not permit PTRANS to result in HEALTH changes. Rather, the effect of a PTRANS is to change LOCATION, which can then enable certain ACTs to occur.

Now, then, the final parse of this sentence would be:



It is now possible to pass such an analysis to an inference program (such as that of Rieger [Rieger, 1975]) that would attempt to make educated guesses

about what would fill in the blanks. Since there is a result causal path from INGEST to HEALTH according to the semantics, and since there is a path along the enable causal path from LOC to INGEST, we can now create a more accurate picture of what was *probably* meant:


I ⟺ PTRANS ← apple ← John, I
PTRANS: p, r; apple: O; John: D
apple ← LOC (John), LOC (I)
John ⟺ INGEST ← apple ← John
INGEST: g, POT; apple: O; John: D
John ⟺ MBUILD → I want to eat the apple
MBUILD: I; I want to eat the apple: O
John ⟺ INGEST ← apple ← John
INGEST: R, p; apple: O; John: D
John ← HEALTH (X + Y), HEALTH (X)
HEALTH: r
(Y > 0)

For the third sentence it can be seen that part of understanding this sentence is being able to react to it in a sort of 'how come' fastion. That is, a hearer would want to know why the first thing should possibly cause the second. We might be able to trace this from the syntax and semantics of causals, but then we might not. Our proposal is that the syntax and semantics of causals should be used to set up as much as can be figured out about a situation, and then to infer the rest. At some point this all must stop, however. Thus, we allow for a simplification that includes an unspecified causal (⇑L) for one thing leading to another in a yet undetermined manner. This lead-to causation is purely bogus and is used only when something better cannot be gotten. It should be clear that in most of our previous use of causals in other papers and in most everyone else's, 'lead to' causation is what is being used. We wish to make clear that this is an abbreviation and should only be used as such.

It is also true that we shall need more abbreviations for the complications arising from trying to write these complicated diagrams and from trying to use them in computer programs. Thus, in the above example, it would be simplest to write ⇑I R r for the causal between the POT-ACT INGEST and the result of the real ACT INGEST (HEALTH change). This abbreviation is always expandable and is just a shorthand.

Thus, we put ourselves in the ambivalent situation of creating a strict syntax for conceptual representation of causality and then only sometimes using it. For a practical usage of causality chains, it is necessary to create abbreviations but to know that these are abbreviations and thus to have the apparatus available to expand them when this is desired. We use the following for example:

ER – enable plus reason causation. This denotes that a POT-ACT was thought about and did occur.

rE – result plus enable causation. This denotes that some state was resulted which enabled some new ACT.

IR – mental causation plus reason causation. This indicates that an ACT was thought about and used as a reason for another ACT.

SCRIPTS

We view the process of understanding as the fitting in of new information into a previously organized view of the world. Input sentences (like input words in intra-sentence analysis) set up expectations about what is likely to follow in the text. These expectations characterize the world knowledge that pertains to a given situation.

The concept of a script, as we shall use it here, is a structure that is made up of slots and requirements on what can fill those slots. The structure is an interconnected whole, and what is in one slot affects what can be in another. The entire structure is a unit that describes a situation as a whole and makes sense to the user of that script, in this case the language understander.

Whenever a story is understood, inferences must be made that will connect each input conceptualization to those that relate to it in the story. This connecting process is difficult and dependent upon the making of inferences to tie together seemingly unrelated pieces of text. However, it is a process that can be facilitated tremendously by the use of scripts.

A script is a giant causal chain of conceptualizations that have been known to occur in that order many times before. Scripts can be called up from memory by various words in the correct context, by visual inputs, or by expectations generated by inferences. What a script does is to set up expectations about events that are likely to follow in a given situation. These events can be predicted because they have occurred in precisely this fashion before. Scripts are associated then with static everyday events such as restaurants, birthday parties, classrooms, bus riding, theater going, and so on. A simple test for when a script is helpful we call the *reference test*. The reference test consists of trying to introduce a new object by use of 'the' and seeing whether the text makes sense. Some examples:

John went to a restaurant.
He asked the waitress to tell the chef to cook him a hot dog.

John went to a restaurant.
He asked the bus driver to talk to the midget.

John went to a birthday party.
First Bill opened the presents and then they ate the cake.

John went to a birthday party.
He asked the waitress to tell the chef to cook him a hot dog.

Every 'the' in the first and third paragraphs makes perfect sense because a script has been introduced and the objects prefaced by 'the' have all been implicitly referenced by the script. In English it is usually incorrect to preface an object by 'the' unless it has been referred to before. It is all right here because the script effectively 'says' that these items were present before they were actually mentioned. When these rules are violated, as they were in the second and fourth paragraphs, it makes the hearer uneasy and he is forced to fill in the missing sentence that introduces these objects by himself. Thus in the fourth paragraph we are surprised that a waitress and chef were present at the party, but we can cope with this information by augmenting our 'birthday party' script to be a 'very fancy party' script.

We are saying that new inputs are handled by hearers in terms of the old knowledge that is lying around about the words and participants in a situation. More important, we are saying that we would expect that, since a script is called up to interpret new inputs and the script contains 'blank events' (events that are expected but not necessarily explicitly mentioned), then hearers will infer the actual existence of these blank events and will often confuse what they have inferred with what they have actually been told. Consider the following two paragraphs:

John went into a restaurant and ordered a hamburger. He enjoyed it and left the waitress a big tip.

John got on the bus to go to Grant Avenue. The driver wouldn't make change for him but an old lady helped him out.

We would expect that people would have no trouble answering the questions 'Who served John the hamburger?' and 'Did John pay the fare?' Yet neither of these events was explicitly in the stories. Rather, they were transmitted implicitly by the scripts attached to those stories. We would claim that scripts take over the understanding task in everyday situations to such a degree as to blur the distinction between what one hears and what one infers.

The general form of a script is a causal chain that connects the expected events of a given situation. A restaurant script might have in it a set of different scenes, each dealing with the kinds of interactions that take place in a restaurant. For example, we might have an 'enter' scene, followed by a 'sitdown' scene, followed by an 'order' scene, followed by an 'eat' scene, followed by a 'pay' scene, and so on. There would be many possible variations that such scenes could have. One possible variety of the 'order' scene might be

MTRANS desire for a menu to waitress
ATRANS menu to customer
MTRANS read menu
MBUILD decide on order
MTRANS order to waitress

Of course, possible variations would have to be built into such a scene. The enabling conditions for each act would have to be checked and if found to be unsatisfied would have to be rectified. (For example, if the waitress isn't near you she will have to be signaled in order to MTRANS the order.)

Such a script is useful because it fills in the blanks in our understanding. (This is similar to the Frames idea suggested by Minsky [Minsky, 1974]). So if we are told that John sat down and later we find that he ate, we can assume that the 'order' scene took place. Alternatively, if we hear that John asked the waitress for the menu and then ordered a steak, we would want to assume that he read the menu, that a steak was on it, and that his order was transmitted to the waitress. All these pieces of information come from the 'order' scene and would be complicated to figure out without such a piece of scriptlike information.

Plans

We might ask where scripts come from. One answer is that scripts are simply plans that have been used a lot.

Plans are sets of information that are attached to the various goals that people have. Some goals we can assume people have are to eat, to be warm, to possess certain things, to have respect, to have power, to know things, to love, and so on. Attached to each of these goals are methods known to the person that are useful for achieving the goals. These methods are possibly explicit sequences of actions that will achieve these goals together with the conditions necessary for the performance of those actions. Or they may be general, only partly formed action lists that have crucial parts either unknown or unachievable that prevent their being put into effect.

However, whether or not a person is capable of achieving a certain goal has little to do with whether he can understand sentences that make use of such goal-achieving or planful knowledge. Consider the following sequences:

John wanted to become chairman of the department. He went and got some arsenic.

John wanted to be chairman of the department. He invited the Dean over for dinner.

John was lonely. He thought if he could find a cab the driver might be able to help him out.

People have general knowledge about plans. They know that to achieve the power goal one has several options. You can do favors for people currently in power, get powerful people to like you, or get rid of the competition, as three (possibly interrelated) options. People use this information to interpret other people's behavior. If you don't have the information that pertains to someone's plan then it is impossible to interpret what you perceive. Specialized knowledge about what is being referred to is needed for the third paragraph above. In the absence of this knowledge about one type of plan, this paragraph is incom-

prehensible.

To give a general idea about how plans work, consider an anthropomorphic bear. This bear lives in a bear world but he and the animals he interacts with can talk. Such a bear has some simple goals: to satiate his hunger, his sex drive, and his need for rest and to preserve his health. Suppose our bear is hungry. He must get some food. Having food in his control will enable the desired INGEST act. The bear uses the plan GET(X) where X is food. GET(X) is a plan that will, he hopes, translate into a sequence of acts that the bear can perform that will result in the desired state (CONTROL(food)), which enables the INGEST. GET(X) is rewritten into two subplans (FIND(X), which will enable TAKE(X). FIND(X) requires knowing where X is located, so this is translated into ΔKNOW (LOC(X)). (Here we borrow the change of state notation called deltacts [Abelson, 1975a]).

The completion of the change of knowledge will enable a PTRANS to the new location, so

$$\text{FIND(LOC(X))} = \Delta\text{KNOW(LOC(X))} + \text{PTRANS(TO LOC(X))}.$$

Many possible plans come from ΔKNOW. If someone else might know, the plan called ASK might be used. The ASK plan consists of the act to be done (an MTRANS of the question 'Where is the food?') and the preconditions that must be satisfied in order to make that plan have a positive outcome. For example, in order to ask, the locations of the asker and the asked must be the same (in the bear world). In addition, the asked must want to convey the answer. The ASK plan is responsible for telling the user how to overcome any unsatisfied preconditions. For example, if the asked doesn't wish to tell, then the asker may wish to THREATEN. The result of the ASK plan is that the asker now knows where the food is. This enables a PTRANS and now the problem of TAKING the food must be met. This comes under ΔCONTROL(X). Under ΔCONTROL(X) there are many possible plans, one of which is BARGAIN OBJECT. If this plan is successful, then control is gotten and the INGEST can take place.

Planning sequences such as that above can be used for both telling and understanding stories. Using the structure and choices above, a generated story might be:

> Joe Bear was hungry. He went to Irving Bird and asked him whether he knew of any bees' nests. Irving said he wouldn't tell him, so Joe threatened to bust him in the beak. Irving said there was a nest two trees down. Joe Bear went there and offered the bees a bunch of flowers in exchange for their honey. The bees agreed and Joe sat down and ate.

The structures above could be used for understanding sequences such as:

> Joe Bear was hungry.
> He found Irving Bird.

Where it was known that birds were not sources of food for bears it could be determined that Irving might be a source of information.

Planning structures are an integral part of memory processes. People know how others handle the world and thus can understand pieces of a plan that they are told about. The theoretical entities that are part of these plans (such as ASK and THREATEN) are simply names of possible sequences of events that will yield the desired result. (They should not be confused with the conceptual representation of the meaning of these words, which is something apart from the intention of an act and its place within a plan. You needn't use the words 'ask' or 'threaten' in order to ASK or THREATEN. The conceptual entities underlying these and other words call up plans that might have been intended.)

When a plan is used often enough it becomes a script. What are scripts for some people are not necessarily scripts for others. What is important here is that language understanding simply cannot take place in the absence of the knowledge that is in memory. Higher level structures such as plans and scripts are just glorified inference techniques. The basis of understanding is the assignment of new inputs to previously stored episodes in memory that will make sense of them. If relevant knowledge is found in memory, understanding can be achieved. Otherwise it cannot be.

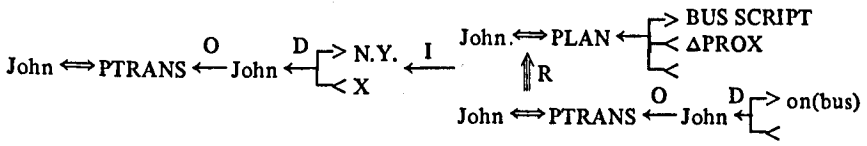
REPRESENTATION OF STORIES

Now let us reconsider the problem of the representation of sentences with respect to their place in a story. Consider the sentence

John went to New York by bus.

This sentence refers to a PTRANS from a sentence point of view, and the bus script called by ΔPROX from the point of view of stories. Should the entire bus script be instrument for the PTRANS here? The issue is entirely representational since the question is not whether John got on the bus or waited for it or whether there was a bus driver, all of which were certainly true. The question is whether these items ought to be part of the representation.

Certainly what must be present as a representation of this sentence is at least the initial PTRANS to New York and probably an instrumental PTRANS to on a bus. This PTRANS to on a bus calls up the BUS SCRIPT. What must be in a representation are the conceptual elements of which we are certain along with a ready access to those element that are somewhat less certain or detail-like that we can expand and peruse when the occasion arises (i.e., when a question is asked). So we propose as the analysis of this sentence:



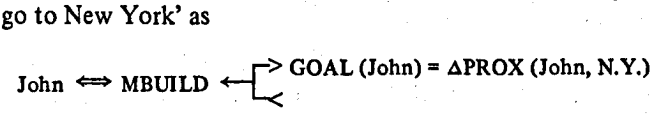
The above is intended to show that John accomplished his PTRANS by means of getting on a bus which was the first step in a PLAN of his. His plan was to satisfy the goal ΔPROX by means of using the BUS SCRIPT.

We are creating a new primitive ACT–PLAN which has two types of input objects–the goal (a deltact often) and some external circumstances relating to the selection of a plan that will realize that goal. (For example, poverty or not knowing how to drive, might be reasons for selecting BUS SCRIPT here). The output of PLAN is the particular plan selected.

BUS-SCRIPT is the name of such a plan. It and ΔPROX are the first non-primitive elements we have allowed in a Conceptual Dependency representation. For years, we have argued that 'sweat', 'dance', 'kiss' and other complicated concepts would have to be broken down into primitive elements. Other researchers have argued that this is too complicated to do and work with, but we have felt there was little choice, given that concepts like 'perspire,' 'foxtrot' and 'embrace' etc. all need reference to the similar elements that underlie them all. So here we are creating non-primitive concepts. What is the difference?

One difference is that these non-primitives do not come directly from words. Rather, they are names of organized bits of knowledge. Often we need to know what knowledge group has been referenced in order to have access to it later on. We may access it when we are asked a question, in which case we can leave the non-primitive unexpanded until that time. However, if some specific piece of the unexpanded plan is mentioned, it must be expanded to accommodate it. This would occur in a story when the first line introduced a plan or script and the second line made explicit some piece of it.

The new primitive ACT PLAN is responsible for PLANning to do something. It serves to relate bodies of knowledge together so as to make explicit the relationship between a goal and the plan to realize that goal. We define a goal as anything that somebody wants. Consequently there can be an infinite possibility of goals for any person. However, there exists a standard set of goals that we can safely assume all people have from time to time. Some of these goals are very low level, like ΔPROX. That is, ΔPROX probably exists only as a subgoal in the realization of some higher level goal. The cognizing or MBUILDing of a goal leads to (and is often the instrument of) a PLAN. Thus we represent 'John wanted to go to New York' as



This is just a shorthand for the old 'going to New York would cause him to be happy'.

Consider the problem of possible sentences following 'John went to New York by bus'.

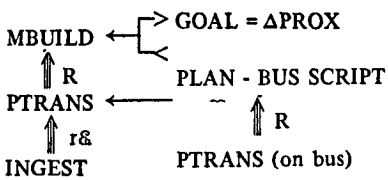
1) After he got there he had some cheese cake.
2) First, he bought a ticket.

Sentence 1) represents the class of sentences that continue the causal chain set up by the PTRANS to New York. The PTRANS results in a location change which enables a PTRANS to wherever they sell cheese cake. Thus, a causal chain can be built according to the rules in Schank (Schank, 1973, 1975) which indicates that this story and the analysis of it is okay.

But for 2) a different problem exists. 2) does not follow in a causal chain from the initial sentence. John's going to New York did not result in or enable 2). Rather 2) is an elaboration of a detail of the initial sentence. The question is how to represent this. The answer must rely on the notion of a GOAL within the ACT PLAN. It is not possible to elaborate upon a description without essentially spelling out the details of a plan to achieve a goal. However, in order to connect 2) to the ΔPROX in the PLAN ACT, we must distinguish between the PLANning of a ΔPROX and the doing of those ACTs that are in the PLAN. That is, there is a difference between thinking about something and doing it.

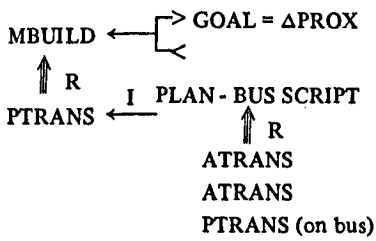
So what we are saying is that sentence 2) is represented twice in the instrument of the main PTRANS. It appears once as the first step in the ΔPROX that is part of the PLAN ACT. It appears in its physical realization as the first ACT caused by this PLAN as reason. Thus the representations of the two stories are as follows:

John went to New York by bus.
After he got there he had some cheesecake.



Here, we indicate the desire to go to New York (MBUILD) was a reason for going (PTRANS) which was done by PLANning a Bus Script, the first step of which we are sure was getting on the bus (PTRANS). The main PTRANS resulted in a location change which enabled an INGEST to a restaurant (abbreviated as *r*&). Hidden in the *r*& is perhaps another PTRANS to a restaurant which we could infer later if need be.

For sentence 2 the representation is:



Here we have that the MBUILD of the goal was the reason for the PTRANS to New York which was done by PLANning it out and executing the steps of getting a ticket and getting on the bus. Any continued description of events connected with the bus script would appear as instrument to the main PTRANS.

This is a major theoretical point and it is worth going into it in detail. We have stated that conceptualizations are tied together by causal chains. Since every ACT has a set of instuments and one of the main causal chain pieces is *enable* causation, there has been some confusion between causality and instrumentality. This confusion existed primarily because of the somewhat artificial problem of analyzing sentences in isolation.

We can see from the above that what is the case is that instruments are a *fuller expansion* of the ACT that they are instrumental to. With respect to causal chains they replace the main ACT that they are instrumental to. The purpose of the main ACT then is to give a handle on the large amount of information in the instrumental chain. We now explain.

From the point of view of causal chains, we wish to connect together every event that results in a state which enables an event and so on. It was our intent (Schank, 1973 and 1975) to connect only those items that could actually have caused each other. Thus, PTRANS can't cause INGEST, it must result in a location change which is near food, which together with some other conditions can enable an INGEST. Suppose the sentence preceding our story had been "John was sitting at home in New Haven when he decided to go to New York and get some cheese cake." This sentence would have filled out the decision process in the initial MBUILD as well as filling in the initial condition and starting point for the bus trip. However, to go from sitting in a New Haven house to a place in New York is a complicated process. By our own rule we would have to specify all the enablements and results in order to enable (from the initial sitting condition) the PTRANS to New York. Now in fact, nothing we could do could enable this PTRANS to New York that would not include specifying at least the entire bus script. What is actually the case is that the entire instrumental chain (including the PLAN and the bus script) should complete the correct causal chain between the initial position (in New Haven) and the final position (in New York). That is, we seem to not need PTRANS at all, rather we just replace it with its instrumental realization.

So shall we replace the PTRANS here (and perhaps all PTRANSes and other such global actions)?

To answer this we must consider what it is we are trying to do. Our aims are two. First, we wish to build a computer program that can understand reasonably complex stories. Second, we wish to come up with a good theory of human processing of such stories.

Consider, for a moment, a human story understander. When a person reads a 300 page novel he does not (unless he is very unusual) remember all the conceptualizations stated in the story in the form of a giant causal chain. Rather he remembers the gist of the book. Maybe 5 or 10 pages of summary could be

extracted from him after reading the book. Previously we have said that Conceptual Dependency Theory will account for memory for gist of sentences. But it cannot be seriously proposed that this is all that is needed for gist of long and complex stories. Some other explanation must be given.

One small part of this explanation lies in the above problem. The concept PTRANS is used as an information organizer. Its presence says something about the goals and intentions of the actor and the overall results and enabling conditions that are important to the story. Without a main PTRANS here, we would be unable to separate the importance of the step onto the bus from the arrival in New York. The main PTRANS serves to focus that importance. It is an abstraction that names the entire sequence. It thus can be remembered that way. It allows for the hearer to concentrate on the main flow and retrace the details later (here, knowing there was a PTRANS allows for tracing what instruments could have been chosen, and recalling the bus, remembering the purchase of the ticket).

We are saying that this is what happens in an example of large text understanding. In a three line story, the process is probably quite different. But embedded in 10,000 kinds of text, such organizational schemes make it possible to remember at all and not get flooded by a morass of conceptualizations.

In a recent experiment, Abelson (Abelson, 1975b) showed that people remember stories better when they are asked to take some particular point of view (of one of the participants or of an observer in a particular place), and that what they remember is contingent on which point of view they had. The ramifications of this experiment for a theory of language understanding have to be that when people have a clue of what to forget they do better at remembering. In other words, good forgetting is the key to remembering. Likewise, if we want to build programs that remember, we had best teach them how to forget. One method of forgetting is simply not noticing levels of detail that are there. This can be done by treating the instruments for an action at a different level of detail than the main ACTs that they explain. When looking at a story at one level of detail we would not see the level of detail underneath it unless specifically called upon to do so (e.g., to answer the question 'Did he pay money to get to N.Y.?').

The principle here is a very powerful and important one. It has ramifications at other levels of understanding as well. Before we get into these, however, we shall attempt to finish our discussion of the representation of stories using scripts and plans.

What we are saying then is that the primitive ACTs essentially perform two functions. First, they represent the concept of action that was stated in a sentence. Second, some of the primitive ACTs, in particular the TRANS ACTs, also serve as names of larger pieces of information. 'John's PTRANS to New York' can be considered both the conceptual realization of a sentence such as 'John went to New York' and the handle by which all of the events that took place on the trip itself can be organized.

Stories are connected in memory of means of causal chains. We are saying now that there are many kinds of chaining. For example, a story might be represented at the highest level as a PTRANS to New York, an INGEST of cheese cake and a PTRANS back to New Haven. But intermediate between these ACTs would be a lot of information. This information is about the actual events that made up the causal chain between the three principal ACTs of the story.

The instruments themselves are causal chains which could also have main parts and subparts. In order to get to the lowest level of detail might require going through several subprincipal ACTs (the fight in the subway on the way to the restaurant.)

At the end of the lowest level of instrumentality would provide a complete causal chain. This chain would never be looked at all at once, but rather the various subhandles would tell the story at various levels of detail.

For example, suppose we had the following story:

> John wanted some cheesecake. He decided to go to New York. He went to New York by bus. On the bus he met a nice old lady who he talked to about the prices in the supermarket. When he left the bus he thanked the driver for the ride and found the subway to go to Lindy's. On the subway he was reading the ads when suddenly he was robbed. He wasn't hurt though and he got off the train and entered Lindy's and had his cheesecake. When the check came, he said he couldn't pay and was told he would have to wash dishes. Later he went back to New Haven.

This story is represented in the new format as:

PTRANS (to N.Y.)	ATRANS (ticket)	
	PTRANS (on bus)	MOVE (foot)
PTRANS (to restaurant)	PTRANS (to on subway)	
	ATTEND (look at fight)	MTRANS (fare)
	ATRANS (get robbed	MTRANS (talk to robber)
	PTRANS (to restaurant)	
INGEST (cheesecake)	not ATRANS (no money to pay check)	
	DO cause CLEAN (wash dishes)	
	PTRANS (to bus station)	
PTRANS (to New Haven)		

Obviously there is more to marking the gist of a story than looking at the principal ACTs without noticing instruments. It is also necessary to mark unusualness at any place that it occurs, so that it is not ignored. The gist of the above story would probably have to mention the robbery and its consequence of

having to wash dishes. This event is more important than the general chain of events.

The point to be made here is this. People learn how to ignore details. Scripts can be added into stories representation and understanding by only looking at the important parts and keeping the rest under the surface. Similarly, any event can be handled by looking at the principal ACT that names the sequence and by looking at the sequence underlying it when necessary.

The process of understanding then, is one of constructing a list of pointers (primitive ACTs) into a flow of events. The list of principal primitive ACTs that is thus constructed is the 'gist' of the story minus any problems that come up in the normal flow of events. Thus, the 'gist' of the above story could be 'John went to New York for cheese cake and then returned home' (PTRANS, INGEST, PTRANS). This gist will work whenever no unusual or problem events occur in a story. But, if a story has these unusual events (as do most interesting stories), these too would be remembered. Thus there must be more than just a straight conceptual dependency causal chain as a representation of a story. There must be an addition to the record of the flow of events (indexed by the principal ACTs as pointer into the causal chain).

This addition we call the WEIRD LIST. The WEIRD LIST is a combination of conceptualizations that were found to be weird by various metrics, together with the belief structures that caused them to be understood as weird. Events on a weird list are given a special place so that they can be tied to their consequences as exemplified later in a story. Thus, here, getting robbed is weird (not on the script for subway riding) so it is placed on the weird list. Its consequence is a lack of money and now we expect a request (or demon) to look for any new events whose enabling conditions require money. 'Going to a restaurant' is one of those and it gets tied to the weird list. This is done by the standard means of causal chaining, triggered in this case by the request for money-needing events. That is, an unenable causal connection is established between 'no money' (the result of the ATRANS in 'rob') and the necessary condition (of money) for the ATRANS in the restaurant script.

Immediately, a problem is created and this problem is put on the weird list too. The problem is 'What will happen next in the restaurant script as a result of no ATRANS?' This problem and its resolution are put on the weird list too regardless of the resolution ('He found money in this shoe' would serve as a resolution too and would appear on the weird list). A problem being resolved is even more weird and would be specifically marked that way on the weird list. (Paying for the cheesecake as if nothing had happened would be an example).

The end result of the story understanding process would thus be: (1) A complete causal chain of conceptualizations connected at the level of the principal primitive ACTs with instrumentality available as necessary and (2) a weird list tabulating problems and their resolution.

The process of summarization would thus be done automatically by consultation of the principal ACTs and the weird list. Such a summary would be:

John went to New York for some cheesecake. He got robbed on the subway and had to wash dishes in the restaurant when he couldn't pay for the cheesecake. Then he went home.

The serious question here is, of course, how is this all to be computed? The answer is that as long as scripts are involved (and we aver that they are involved in a tremendous part of understanding (see also Abelson [Abelson, 1975b]); the answer is relatively easy. First, scripts are computed and stored as instruments of principal actions (based on the goals of the actor (the deltacts and others). As long as scripts are referenced they can be reconstructed by the hearer and are thus easily forgotten. Thus, the process of understanding, where scripts are involved, is one of ascertaining what script is being referred to and establishing therefore what the actors' goals are. The scripts are thus 'forgotten' in the sense that they are stored elsewhere than in the main causal chain. The exceptions to this are events that are not predicted by the script. These events are placed within the instrument but are also added to the weird list. The method for doing this is simply lack of correspondence with predicted script parts. When such disjointness is noted, it is put on the weird list and from that point the computed probable consequences of the weird event are 'kept in mind' (as requests) whenever a new script is entered. New scripts are, from that point, expected to have deviances if their entry conditions were affected by the weird event. They are thus paid closer attention to (kept within the main causal chain).

One of the major issues in Artificial Intelligence research must be the creation of a theory of forgetting. It simply is not possible to assume that people do, or that machines should, remember everything they encounter. In listening to a speaker, reading a book, or engaging in a conversation, people could not possibly remember everything they are told verbatim. In attempting to get the gist of a sequence, they must employ what we call *forgetting heuristics*. As part of these forgetting heuristics are heuristics that search out items of major importance. The selection of these major items is the key to forgetting. We don't really wish to assert that people couldn't possibly remember everything they hear. Rather we wish to find a procedure that will let us see only the major items, yet also find, with some difficulty, the thoughts or statements that underlie them, and the ideas that underlie those, and so on.

Thus, the key to understanding must be, in order to facilitate search among what has been understood, an organization of the new information, in such a fashion as to seem to forget the unimportant material and to highlight the important material. Forgetting heuristics must do this for us. So the first task before us is to establish what the most significant items in a text are likely to be, and then to establish the heuristics which will extract and remember exactly those items.

REFERENCES

Abelson, R.P. (1975a) Concepts for representing mundane reality in plans. *Representation and Understanding: Studies in Cognitive Science* (eds. Bobrow, D. and Collins, A.), Academic Press.

Abelson, R.P. (1975b) Does a story understander need a point of view? *Using Knowledge to Understand.* (eds. Schank, R. and Nash-Webber, B.) Proceedings of the Conference on Theoretical Issues in Natural Language Processing.

Goldman, N. (1975) Conceptual generation. *Conceptual Information Processing,* (ed. Schank, R.), North Holland Publishing, Amsterdam.

Katz, J. (1967) Recent issues in semantic theory. *Foundations of Language,* III.

Minsky, M. (1974) Frame-systems. *AI memo*, MIT.

Rieger, C. (1975) Conceptual memory. *Conceptual Information Processing,* (ed. Schank, R.) North Holland Publishing, Amsterdam.

Riesbeck, C. (1975) Conceptual Analysis. *Conceptual Information Processing,* (ed. Schank, R.) North Holland, Publishing, Amsterdam.

Simmons, and Slocum (1972) Generating English discourse from semantic networks. *Comm. of the ACM.* 15, 10.

Schank, R. (1972) Conceptual dependency: a theory of natural language understanding. *Cognitive Psychology,* 3, 4.

Schank, R. (1973) Causality and reasoning. *Technical Report No. 1.* Instituto per gli studi Semantici e Cognitivi. Castagnola, Switzerland.

Schank, R. (1975) The structure of episodes in memory. *Representation and Understanding: Studies in Cognitive Sciences*, (eds. Bobrow, D. and Collins, A) Academic Press.

Schank, R., Tesler, L. and Weber, S. (1970) Spinoza II: Conceptual Case-Based Natural Language Analysis. *AI Memo, No. AIM-109,* Computer Science Department, Stanford University, Stanford, California.

Winograd, T. (1971) Procedures as Representation for Data in a Computer Program for Understanding Natural Language. *TR-84,* MIT Project MAC.

SUBJECT AND NAME INDEX

[Page numbers are in roman type, chapter numbers in italics. Names only appear which are in no sense bibliographic references; otherwise they are to be found in the author index.]

A* search algorithm, 60-1
ABSET language, 209
ABSYS language, 269
accommodation, in Piagetian psychology, 378-80, 383, 388
ACT system, 322
actors, 426-7
addlists, add-delete lists, 120-2, 130-3
admissibility, in heuristic search, 60-1, 67-8
advice, 151, 157-68, 176, 219
advice theory, *8*
advising, 237, 239, 260, 262, 264
agitation, in a chess position, 143
Alexander, C. H. O'D., 149
ALGOL language,
  ALGOL 60, 269, 279-82, 442
  ALGOL 68, 226
  ALGOL-like, 148, 229, 254, 271, 286-7, 306, 316-8, 323
  list-processing and other extensions, 278-83
ambiguity
  in chromosome classification, 455
  in language text, 541, 545, 577-9, 586, 590
  in scene analysis, 484-5, 493
Amsterdam (Vrije Universiteit), 55
Andelfingen cemetry, 438, 447-8, 450-1
Anderson, B., 404
antecedent theorems (in PLANNER-like systems), 104, 533-4
AP/1 language, 122
APL language, 225-6
AQVAL/1 system, 175, 177-8, 184
archaeology, *21*
Artificial Intelligence (AI),
  and cognitive psychology, 329
  at human level, 172
  languages, 105-121, 209-12, 266
  lectures by Winograd, 241
  Minsky's view, 535
  and problem of relevance, 512
  programs, 210, 221, 306-7, 321, 445
  researchers, 7, 56, 148, 231, 243
  in teaching, 7, *18, 19, 20*
assignment in programming, 106-7, 125, 230, 256, 259, 337
Awerbach, J., 168
axioms,
  field axioms, 44
  frame axioms, 120, 133, 211
  in geometry, 11, 18-27
  in resolution theorem-proving, 73-4, 269-72
  in probability theory, 139-48

backtracking,
  in computer chess, 142-3
  in PLANNER-based systems, 122, 220, 549
  in PROLOG, 271
  in scene-analysis, 526, 532, 534
  in standard search methods, 70
backward chaining, backchaining,
  in geometrical proof, 18, 21, 25
  in production systems, 319
  in program validation, 37, 40, 51
BASIC language, 226
Bayesian statistics, 147, 472
beliefs, 140-1, 617
Bennett, C., 393
binding of names to values, *13*, 230, 314, 561
blocks program, blocks world, 95-8, 113, 211, 579
Blomme, R., 393
branch-and-bound method, 60, 68, 366
British Columbia, University of, 510
British Museum algorithm, 73
Büchi, cited by D. E. Knuth, 152
bugs,
  in programs, 132
  in production systems, 327
  as sources of insight, 510
  in teaching, 385

California, University of (at Santa Cruz), 8, 55, 73, 86, 475, 493
Caratheodory's Theorem, 500
Castagnola (Institute for Semantic and Cognitive Studies), 541
causal chain, 614, 616-7
causality, 597-606
cemetry sites, *21*
centromeric index, 455, 457, 467, 470
chess
  complexity, 57, 153-67

computer chess, 7, 366-7
end-games, *10, 11,* 69, 149, 159
puzzles, 91-2
Chomsky, 267
chromosomes, *22*
Clowes, M. B., 7
cluster analysis, 440, 451
cognitive dissonance, 383
Colmeraner, A., 283, 292
combinatorial explosion,
in archaeological inference, 445
in heuristic search, 146
in problem-solving, 69, 92
in real-world inference, 544, 549-50
in theorem-proving, 37, 74
COMIT language, 217
complexity,
of chess, 57, 153-67
of control structures, 321
of genetic structures, 206
non-polynomial (NP), 55-57, 66, 70
of problem-domains, *6·2·8,* 155, 166, 271-2, 308, 329
of programs, 224, 243, 317
of proofs, 73, 79, 83-4
of propositions, 171-2
theory, 58, 70, 152
computer-aided instruction, 7, *81-20*
computer vision, 7, *23-25*
computers,
CDC 6600, 45
Cyber 73-2, 390
IBM 704, 19
PDP 6/10, 596
PDP-10, 45
PDP-11/35, 158
Conceptual Dependency, *27*
conceptual structures, 576-7, 580-1
Conniver, 122, 126, 209, 217, 220-1, 296
consequent theorems (in PLANNER-like systems), 104, 325-6, 556
contexts,
in AI languages, 105, 122-4, 129-34, 211-13, 217, 220
in production systems, 315-6, 319
in story-understanding, 572
control structures,
in AI languages, 210, 212, 220-1, 237
in production systems, 302, 306, 314-22, 327, 329
convex hull (of a set), 500
COPY DEMO system, 510, 535
cost-benefit of computations, 160, 164
costs,
of analysis in chess, 141-5
of computation, 59, 66, 147, 464, 470
of paths and tours, 56, 59, 67
of trees, 67-8
courseware, 387-90, 400-4, 409
cracks (in scene analysis), 531-2
crossing numbers (in image analysis), 461-4
cut-sets (in scene analysis), 482-3, *24*

database management, 261-2, 549
data-driven programs and procedures, 231-62, 317, 446
de-bugging,
as analogy for learning, 368, 385
of frame implementations, 567
of LISP programs, 225
by machine, 336, 358
of problem-solvers, 132
of program-synthesis rules, 336
deletelists, 120-2, 130-3
demons,
in AI languages, 218, 236
in language understanding, 551-59, 572-4, 617
in production systems, 309
in archaeological inference, 437, 447-52
DENDRAL system, 303, 306, 309-11, 315-6, 322-5, 445
Dennis, J. R., 398-9
diagrams,
in geometrical proofs, *1*
in program synthesis, 338-41
difficulty of problems, 68, 166-7
Dijkstra, E. W., 281
dual, duality (in scene analysis), *23*
dynamic weighting (in heuristic search), 55, 68-9

Edinburgh,
University of, 151
Western General Hospital, 455
efficiency,
of computations, 7, 278, 303
of control structures, 70, 318
of learning, 364
of matching, 322
of plans, 117
of program constructions, 219
of programs, 279, 357, 444-5
of search algorithms, 55-62, 210
of world-modelling, 125-9, 133
end-games in chess, *10, 11,* 69, 149, 159
entropy (use in pruning search), 145
EPL language, 230
equality, 30-1, 34, 280
error,
in function-evaluation, 167
in heuristic search, *3*
in student performance, 384
Essex, University of, 433
evaluation functions,
in game-playing, *7, 12*
in heuristic search, 62-3, 66
evidence, 142, *9*
existential quantification, 25, 105, 219

factor analysis, 433, 451
formal systems, formalisms, formalizations,
actor formalism, 130
of knowledge, 159, 164
in linguistics, 267, 290, 292
in mathematical reasoning, 11, 25
using predicate logic, 130

of probability, 139
in production systems, 306, 311, 318
in puzzle-representation, *5*
Sacerdoti formalism, 132
STRIPS formalism, 125
FORTRAN language, 228, 277
frame axioms, frame problem,
in Merlin, 70
in program synthesis, 100
in resolution-based problem-solvers, 120, 124-5, 128, 133, 211
in relation to QLISP, 218
frames (for knowledge-representation), 445-7, 517, 560-74

game theory, game-theoretical, 141, 153-4, 157
Gaussian sphere, 487, 491
General Problem Solver (GPS), 120, 133, 210
goal-protection in problem-solving, 95-118
goals and sub-goals,
in archaeological interpretation, 442
in chess, 146-7, 193
in complex programs, 224, 266
in geometrical proof, 14-18
in heuristic search, 59, 210-11
in PLANNER, 219-20
in problem-solving and program synthesis, *6*
in program validation, 26, 31, 34-7, 40
Graph Traverser (GT) program, 60, 66
Green, C.C., 270

HACKER problem-solver, 117, 423
Hallstatt cemetry, 435, 451
Hamilton path problem, 55, 57
"handles" in programs, 251, 257, 262
hash-coding, 241
HBASE system, 122
HEARSAY II system, 445-6
Held-Karp bounding function, 55
Herbrand base, interpretation, universe, 273-5
heuristic functions, 58, 62, 65-68
Heuristic Path Algorithm (HPA), *3*
heuristics,
in advice theory, 160
for forgetting, 618
in geometrical proofs, 26
to guide search, *3*
in problem-solving, 141
in production systems, 324, 326
in scene analysis, 519, 521, 528
Hilbert, D., 11
Hoare, C.A.R., 157
Horwood, E., 8
Huffman, D., 7-8

Illinois, University of (at Urbanan-Champaign), 175, 193, 363, 369, 387, 390
implementation,
of the Geometry Machine, 19, 25-6
of learning in chess, 176
of a problem-solver, 115, 118, 120
of production systems, 301
of skeleton models, 129
impossible objects (in scene analysis), 493, 527, 533
induction,
mathematical, 61, 69
scientific, 148, 171
inductive inference, *9-12*
inequalities,
in probability judgments, 140-147
in program validation, *2*
inserts
in production systems, 317
in programs, 237-240, 263
instantiation,
in an actor system, 427
in frame systems, 561-2, 569-72
in geometrical proofs, 15-16, 19, 27-8
in program synthesis, 97
in resolution theorem-proving, 76
INTERLISP system, 230, 236, 245
interpretations,
in archaeology, *21*
in geometry, 11, 27
in language processing, 26-7
of line drawings, 525-34
in logic, 21
procedural, of logic, 283, 285
interpreter,
for ALGOL, 280, 283
for LISP, 262, 325
for PLANNER, 551
for predicate logic, 7
for production rules, 301, 304, 312, 326
for PROLOG, 271

karyotypes, karyotyping, *22*
Kester, T., 8
Kleene, S. C., 267
knowledge
acquisition, (by humans), 329, *18-20;* (by machines), 7, 132, 162, *9-12*, 328, see also under learning systems;
base, 31, 162, 300, 329, 335
engineering, 148
measurement, 7, *7-8*
about programming, *17*
representation, see Table of Contents
sources, 445-7
transfer, 7, *17-20*
about the world, *26-7*
Knuth, D. E., 167
Kowalski, R. A., 7, 269, 283
KRL language, 451

Lasker, Emanual, 149
la Tène culture, 437-8, 447
LEAP language, 209, 213-5

learning systems, 132, 146, 158-9, 166, 312, 327-9
Lighthill, J., 147, 172-3
Linkoeping University, 223
LISP language,
in conceptual programming, 223-30, 234 234-7, 243-4, 256, 260-2
for fact retrieval, 548, 567
in relation to production systems, 303, 322-6
as target of program synthesis, 343, 353, 356-7
LOGO language, 422-5, 428-9
LUSH system (for resolution logic), 271, 283-5, 288, 291-8

McDonald, D., 225, 233, 243, 251
machine intelligence research, 147, 151, 156, 167, 171-2
Machine Intelligence series, 7-8
machine specification (in advice theory), 152, 162
MACLISP language, 233
Malvern (Royal Radar Establishment), 209
MARGIE program, 575, 595-7
Marina Del Rey (Information Sciences Institute), 45
Marseilles, University of, 271
Martingale property, 144
mass spectrometry, 306
Massachusetts Institute of Technology (MIT), 225, 233, 422
mathematics,
mechanization of, 7, *1, 2*
teaching, *18, 19,* 424
strategies in, 11, 69, 74, 140-1, 148
medicine, 306-7, 325
memo-functions, 277
MERLIN system, 70
Meyer, A., and Stockmeyer, L., cited by D. Knuth, 152
Michie, D., 146, 148-9, 172, 206
MICROPLANNER language, 296, 552-3, 556-7
mini-procedures, 369-77, 384-5
Minsky, M., 383
models,
in archaeology, 452-3
in chess, *11*
in geometry, 22, 26-8
in logic, 21, 24
in problem-solving, 58, 97, 101-21, 124-34
psychological models, 302, 323, 544-5
object models (in scene analysis), 511-13, 517, 535
world models (in natural language), 602; (in programming languages) 212
monotone property (in heuristic search), 55, 61-2, 67
multiplication of advantage (in chess), 144
multivariate analysis, 440-1, 451-2
Münsingen cemetry, 438, 440-2, 447, 451

MYCIN system, 171, 303, 306, 316-18, 323-7, 330, 445
Nebringen cemetry, 447-8, 451

occlusion (in scene analysis), 510-11, 527, 529, 532, 534
operators (in problem-solving), 120-21, 128, 130, 133
oracles (in worst-case analysis) 62-5

Page, D., 374, 384-5, 388
Papert, S., 383, 423
PASCAL language, 225-6
pattern-directed invocation,
in an actor system, 426
in problem-solving, 97
for controlling demons, 450
for data0driven calls and advising, 236-7
in PLANNER-like systems, 281, 549, 551, 559
pattern-matching,
in an actor system, 426
in relation to binding mechanisms, 212-221
in PLANNER-like systems, 281, 450, 549, 553, 556
in predicate-ligic programming, 280-1, 287,
in production systems, 313-14, 322-3, 325
pattern recognition, 455, 517
patterns (in chess), 184, 203
"Peanuts" characters, 428
perceptions, 365-7
permutations,
in path problems, 56
in sorting problems, 96, 271, 276, 278, 288-90, 338-9, 345
in space-filling problems, 87, 92
perspective (in scene analysis), 513
Piagetian psychology, 363, 378, 388, 409
Plaisted, David, 152
PLANNER language,
for artificial intelligence, 209, 217-220, 281
in relation to demons, 450
for fact-retrieval, 548-51, 557, 559-60
in relation to production-rule selection, 325-6
for representing mathematical knowledge, 25
PLANNER-like systems,
in relation to heuristic search, 70
in relation to resolution logic, 295
planning, *6,* 609-13
PLASMA language, 451
PLATO system (for computer-aided instruction), 363, *19*
PL/I language, 225-6, 267-9
Polin, G., 401
POLY program (for scene analysis), 534
Polya, G., 148
polyhedra (in scene analysis), *23, 24, 25*

POPLER language, 209
pragmatics for resolution logic, 274-5, 283
predicate calculus,
  in a database system, 232, 235
  as basis for problem-solvers, 130, 211, 218
  as a programming language, 7, *15*
  for representing knowledge, 308, 445, 543, 547-51, 559
  theorem-proving, 19-21,
    see also resolution.
Preece, L., 8
Preece, S., 8
Presburger Arithmetic, 33
prime number theorem, 148, 159
probability,
  dynamic probability, 7, 171
  intuitive probability, 139, 148
  logical probability, 139
  physical probability, 148
  subjective probability, 139, 147, 172
  superficial probability, 143
  tautological probability, 148
problem-solvers, problem-solving, 7, *3-6*, 147, 269, 278
production rules, production systems, productions, 293, *16*, 378, 445-6
program generation, program synthesis,
  in conceptual programming, 224, 228-9, 236, 240, 255-6, 264
  by dialogue interaction, *17*
  by inductive inference, *10, 11*
  in predicate-logic programming, 267, 270, 278, 282
program modification, *6*, 237, 247, 263, 315, *17*
program simplification, 338, 357
program validation, program verification, *2*, 99, 119, 270
PROGRAMMAR language, 293-5
programming,
  automatic programming, see program generation
  conceptual programming, *14*
  languages, see Artificial Intelligence languages or under names of languages
  using production systems, *16*
  using resolution logic, 7, *15*
  structured programming, 247, 318
PROLOG language, 271-2, 277, 284, 291, 293, 295-6
puzzles,
  Eight Queens, *5*
  Fifteen-puzzle, 59
  Instant Insanity, *5*
  Soma Cube, *5*
  Tower of Hanoi, 56

QA4 language, 122, 209, 220
QLISP language, 94, 122, 127, 209, 218, 221
Quicksort, 157, 271, 279, 288, 295
quiescence (of chess positions), 143

rationality, 142, 147, *9*
resolution,
  in fact retrieval, 543-4, 547
  in relation to pattern-matching, 217
  in predicate-logic programming, *15*
  in problem-solving, 7, 58, 120, 133
  in program verification, 30, 46, 48
  theorem-proving, *4*
robots, 96, 112, 120, 125-6

SAIL language, 214
Samuel, A. L., 365
scene analysis, 433, *23, 24, 25*
Schank, R., 557
scripts (in language understanding), 560, 575, 607-14, 617-8
search,
  in archaeological interpretation, 442
  in chess, 142-7, 203
  in the Geometry Machine, 21-6, 29
  heuristic search, 7, *3*
  in resolution theorem-proving, *4*, 284-95
SEE program (for scene analysis), 517-527
segmentation,
  in image analysis, 456-7, 460-1, 472
  in scene analysis, 510, 517, 520, 523, 528-9
semantic networks, 445
semantics,
  conceptual semantics, 605-6
  in geometrical proof, 21-3, 26, 28-9
  of natural language, 7, 235, 575, 584
  in production rules, 318, 327
  of programming languages, 336
  of programs, 246
  of resolution logic, 266, 273-4, 283
  of the Travelling Salesman problem, 56
shadows (in scene analysis), 488-9, 531, 534
SHRDLU program (for language understanding), 549
Sickel, S., 8
side-effects, 118, 128, 219, 311, 320, 327
SIMULA 67 language, 225, 231
SIR program (for fact retrieval), 548
Skinner, B. F., 363-4
skolem constants, skolem variables, skolemization, 31, 41, 44, 47-8
Smith, S. G., 397
SNOBOL language, 217, 225
SOLCEM program (for archaeological interpretation), *21*
Solomonoff, R.J., cited by G. Chaitin, 166
sorting,
  problem, 96, 109, 116, 119, 121, 156-7
  programs, *15, 17*
  theorist, 58
Stanford Research Institute (SRI), 94
Stanford University, 300, 335, 595
Sternberg memory scanning tasks, 305
strategies
  adversary strategy, 62, see also oracle
  in mathematics, 11, 69, 74, 140-1, 148
  in PLATO games, 406-9

in problem-solving, *6*, 220, 324
in production systems, 327
in teaching, 384, 402
in theorem-proving, 7, 18, 26, 58, *4*
STRIPS problem-solver, *6-2*, 272
STUDENT program (for question-answering), 321
subgoals, see under goals
super-procedures, 375, 377, 385
surprisals (in advice theory), 163-4
Suzman, P., 171, 172
syntax,
conceptual syntax, 605-6
in geometrical proof, 11, 22-9
in natural language, 575
in picture processing, 525
for predicate logic, 272-3, 283
of production systems, 315, 322, 328
of programs, 246
syntactic strategies in theorem-proving, 73

Tenczor, P., 391-2
Texas, University of (at Austin), 30, 45
theorem-proving by machine, *1*, *2*, *4*, *15*
thresholds, thresholding (in image analysis), 458, 461, 466, 472
topology (in scene analysis), 512-3, 517, 519-20
"trans" concept (in language understanding), 576-8
Travelling salesman problem (TSP), 55-6, 66 66-9
turbulence (of chess positions), 143
Turing, A. M., 218-19, 328
Turing machines, 267, 301
Turnbull, A., 8
TUTOR language, 391
types and typing, *2*, 220, 230, 233, 244-54, 279

ultraintelligent machine (UIM), 173
understanding
natural language, 272, 426, *26-27*
program text, 223-4, 315, 335, *17*
speech, 445
unification,
in pattern-directed invocation, 217, 219, 221
in theorem-proving, 44, 48, 75-6, 272
universal quantification, 19-20, 31, 274
utilities, 140-147, 164
utility index (in games), 203

Virginia Polytechnic Institute and State University, 139, 171, 205

Waltz, D. L., 7
WARPLAN problem-solver, 121
Waterloo, University of, 266
weird list, 617, 618
Western Ontario, University of, 11
Winograd, T., 549
Wittgenstein, L., 139
Woods, W., 7
world,
hypothetical worlds, *6-2-7*, 134
states of the world, 31, 101, 130, 133
worst-case analysis, 55, 62-5

Yale University, 575

AUTHOR INDEX

[Page numbers are in roman type for text references, in italics for bibliography entries. Bold type denotes chapter numbers.]

Abelson, R. P., 560, *574*, 610, 615, 618, *618-9*
Abelson *et al*, 429
Alo, A. V., 67, *70*
Ambler, A. P., *185*
Anderson, J., 301, 318, 322, *331*
Anderson, R., *85*
Anderson, R. H., 525, *535*
Andrews, P. B., *85*
Aristotle, 160-2
Arnold, M., 115
Atwater, T., *93*
Awerbach, J., 175, *184*

Balzer, R. M., 122, *134*
Barrow, H. G., 122, *134*
Barstow, D. R., 106, 109, 117, *135*, 156, **17**, 336, 338, *358-9*
Bartlett, F. C., 517, *535*
Battani, G., *298*
Beckman, L., 231-2, *264*
Bergman, M., 272, 295, *298*
Berliner, H., 139, 171, *173*
Binford, L. R., 453, *453*
Bitzer, D. L., 392, 398, 400, *420*
Bledsoe, W. W., **7**, **2**, 30-4, 36, 38, 44-5, *48*, *85*, 273, *298*
Bobrow, D. G., 70, *71*, 122, *134*, 209, 213, *221*, 269, 272, 281, *298*, 321, *331*, 446, *453*
Boyer, R. S., 30, *48*, *85*, 119, *134*, 273, *298*
Brillhart, J., 157
Brown, T. A., *93*
Bruner, J. S., 384, *385*
Buchanan, B. G., 319, *331*, 445, *453*
Buchanan, J. R., 122, *134*
Buckton, K., 455, *472*
Bundy, A., 30, *48*
Bursky, P., 59, *72*
Burstall, R. M., *48*, 119, *134*
Butler, J. W., *472*
Butler, M. K., *472*

Card, W. I., 145, *149*
Carson, D. F., 74, *85*
Chaitin, G. J., 152, 155, 166, *168*
de Champeaux, D., 66, *71*
Charniak, E., **7**, **26**, 549, 559, 563, 565-6, 574, *574*
Chernev, I., *184*
Chomsky, N., 525, *535*
Clarke, M. R., 168, *168*
Clowes, M. B., 511, 517, 525-7, *535-6*
Cohn-Vossen, H., 487, *492*
Colmerauer, A., 271-2, 290, 293, 295, *298*
Cook, S. A., 56, *71*
Coolidge, J. L., *173*
Cooper, D. C., 33, *48*
Crook, J. F., 147, *149*
Cuneo, R. P., 177, *184*

Dahl, O. J., 237, *264*, 337, *359*
Darlington, J., 109, 119, *134*
Davies, J. M., 209, *221*
Davis, R., **16**, 303, 316, 318-9, 324, 327, 330, *331*, 445, *453*
Davis, R. B., **18**, 363, 365, 369, 372, 374, 384, *385*, **19**, 388, 390, 402, 421
Delong, R., *93*
Derksen, J. A., 30, 44, *48*, 122, 130, *134*, *136*
Dijkstra, E. W., 99, *134*, 337, *359*
Disraeli, 100
Doran, J., 58, *71*, 211, *221*, 21, 433, 440, 442, 452, *453*
Dugdale, S., **19**

Elcock, E. W., *1*, 209, *221*, 269, *298*
Eliot, T. S., 96, 105-6, 119, 130-1, 133-4
Elschlager, R., 106, 109, 117, *135*, 338, *359*
van Emden, M. H., **7**, **15**, 273, *298*, 357, *359*
Enea, H. J., *331*
Erlwanger, S. H., 369, 372, 376, 380-1, *385*
Erman, L. D., 445, *453*
Evans, A., 329, *331*
Evans, T. G., 525, *536*

Fahlman, S., 122, 126, *135*, 446, *453*
Falk, G., 521, 523, *536*
Fay, M. J., 87, *93*
Feigenbaum, E., 73, *85*, 303, 329, *331*
Feldman, J., 73, *85*, 209, 214, *221*
Fikes, R. E., 120, 122, 127, *135*
Fine, R., 175, *184*

Fischer, M. J., 152, 168
Flavell, J. H., 388, *421*
Fletcher, J., *93*
Floyd, R. W., 98, 119, *135*, 329, *331*, 337, *359*
Foster, J. M., 13, 209, *221*, 269, *298*
Gardner, M., *93*
Gelernter, H., 18-19, *29*
Gelperin, D., 75, *85*
Gilmore, P. C., 19, 21, 24, *29*
Glaser, R., 364, *385*
Golden, W., 393-7, *421*
Goldman, N., 595, *619*
Goldstein, I. P., 25, *29*, *429*
Goldstine, H. H., 98, *135*
Good, D. I., 45, *48*
Good, I. J., 7, *139*, 140-5, 147-8, *149*, 9, 171-2, *173-4*, 12, 205, *206*
Gray, P. M. D., 209, *221*
Green, C. C., 106, 109, 117, 120, *135*, 156, 269-70, *298*, 17, 336, 338, 357, *359*
Greenfeld, N. R., 122, *134*
Greenstein, R., 372, 374, *385*
Gregory, R. L., 517, *536*
Guzman, A., 517, *536*

Halmos, P. R., 365, *385*
Hansen, J. R., 18-19, *29*
Haraldson, A., 231-2, *264*
Harburg, E. Y., 95
Hardy, G. H., 159, *168*
Harris, L., 69, *71*
Hart, P., 60, *71*, *85*
Hartley, R. V. L., 145, *149*
Hayes, P., 100, *135*, 271, 298
Hearn, A. C., 44, *48*
Held, M., 67, *71*
Henneman, W. H., 30, *48*, 273, *298*
Hewitt, C. E., 25, *29*, 58, *71*, 97, 104, 112, 122, 130, *135*, 209, *221*, 325, *331*, 426, *429*, 548, *574*
Hilbert, D., 487, *492*
Hilditch, C. J., *472*
Hill, R., 283-4, *298*
Hoare, C. A. R., 99, 119, *135*, 278-9, *299*, 337, *359*
Hodson, F. R., 433, 436-8, 440-2, 452, *453*
Holland, J. H., 206, *206*
Hollander, D. H., *93*
Homer, 122
Hopcroft, J. E., 56, 67, *70*
Horn, A. 283, *299*
Horowitz, E. A., *150*, *184*
Huffman, D. A., 7, 23, 475, *492*, 24, 493, 496, 506, 508, *509*, 525-7, 531, 534, *536*

Jensen, G. M., 178, *184*, 204, *204*

Kac, M., 11, *29*
Kahn, K., 20
Kanoui, H., 271-2, 295, *298-9*
Kant, E., 336, *358*
Karp, R., 57, 67, *71*
Katz, J., 576, *619*
Katz, S. M., 119, *135*
Kaufmann, B., 365, *386*
Kawai, S., *93*
Kay, M. J., 122, *134*
Kestner, J., 66, *72*
Keynes, J. M., 140, *150*
Kibbey, D., 19
King, J. C., 16, 99, *135*
Kirkpatrick, D. G., 62, *71*
Kirsch, R. A., 525, *536*
Kleene, S. C., 272, *299*
Kling, R. E., *85*
Knuth, D. E., 62, *71*, *93*, 151, 157, 167, *168*, 340, 357, *359*
Koestler, A., 363, *385*, 388, *421*
Koopman, B. O., 148, *150*
Kowalski, R. A., 58, *71*, 75, 77, *85*, 104, 130, 130, *135*, 270-1, 273-4, 283, 285, 289-90, 296, *298-9*, 357, *359*
Krämer, W., 447-9, *453*
Kromer, K., 435, *453*
Kuehner, D. G., 75, *85*, 270, *299*
Kuhn, T. S., 388, *421*
Kuipers, B. J., 446, *453*

Larson, J., 177-8, *184*, 195, *204*
Larson, L. C., *93*
Lauder, I., *472*
Lederberg, J., 445, *453*
Ledley, R. S. 525, *536*
Lee, R. C. T., 120, *136*
Lemoine, E., 172, *174*
Lenat, D. B., 106, 109, 117, *135*, 338, *359*
Lenat, D. L., 330, *331*
Lesser, R. L., 330, *331*
Lesser, V. R., 445, *453*
Levan, A., 455, *472*
Levitt, K. N., 30, *48*, 112, *136*
Lighthill, J., 69, *71*, 173
Lin, S., 69, *71*
London, R. L., 45, *48*
Loveland, D. W., 18-19, *29*, 75, *85*, 270, *299*
Luckham, D. C., *85*, 122, *134*
Lumsdaine, A. A., 364, *385*

McCarthy, J., 100, *135*
McCune, B. P., 106, 109, 117, *135*, 338, *359*
McDermott, D. V., 122, 130, *135*, 209, *221*
McGregor, J. J., 209, *221*
McKeeman, W. M., 5, 87, *93*
Mackworth, A. K., 25, 532, 534-5, *536*
Mann, W. C., 122, *134*
Manna, Z., *48*, 96, 99, 109, 112, 118-19, 130-1, *135*
Martin, W. A., 357, *359*
Martin-Kilcher, S., 438, 442, *453*
Mason, D., *472*
Meeus, J., *93*
Meloni, H., *298*
Mendelsohn, M. L., 458, *472*
Mendelson, E., 73, *85*

Michalski, R. S., 171, 10, 176-8, *184*, 195, 202, *204*, 205
Michie, D., 58, 59, 66, *71*, 144, 149, 8, 153, 159, 167, *168*, *185*, 206, 211, *221*, 273, 277, *299*, 511, *536*
Minsky, M., 58, *71*, 142, *150*, 278, *299*, 365, 369, *385*, 446, *453*, 517-18, 521, 525, 535, *536*, 560, *574*, 609, *619*
Moberg, C. A., 436, *453*
Moise, E. E., 365, *385*
Moore, J S., 70, *71*, *85*, 119, *134*, *135*
Moran, T. P., 302, 315, 320, 322, 328, *331*
Morganstern, O., 141, *150*
Morrison, M., 157
Moses, J., 44, *48*
Mott-Smith, G., *150*
Murray, A. N. 209, *221*

Nagel, R. N., 458, *472*
Narasimhan, R., 525, *536*
Naur, P., 98, *135*, 282, *299*
Negri, P., 10,,11, 202
Ness, S., *48*
von Neumann, J., 98, *135*, 141, *150*
Newborn, M., *185*
Newell, A., 70, *71*, 120, *135*, 210-11, *221-2*, 301-2, 305-7, 317, 321, *329*, *331*
Newman, C., 203, *204*
Nilsson, N. J., 59-60, 69, *71*, 120, *135*, 273, 284, *299*, 442, *453*
Noshita, K., *93*

O'Gorman, F., 511, *536*
Oskarsson, Ö., 231-2, *264*

Papert, S., 58, *71*, 278, *299*, 365, 369, *385*, *429*, 518, 521, *536*
Pasero, R., 271-2, 295, *298-9*
Paton, K., *472*
Peebles, C. S., 436, *453*
Pennello, T. J., 87, *93*
Penrose, L. S., 531, *536*
Penrose, R., 531, *536*,
Pierce, C. S., 142, *150*
Piranian, G., 365, *386*
Pitrat, J., 176, *185*, 203, *204*
Pohl, I., 3, 58-60, 62-6, 69, *71-2*
Polanyi, M., 363, *386*, 388, *421*
Polya, G., 141, 148, *150*
Post, E., 300, *331*
Pressey, S., 363, *364*
Prewitt, J. M. S., 458, *472*

Ramsey, F. P., 140, *150*
Randsborg, 436, *454*
Raphael, B., 60, 70, *71*, *85*, 122, *134*, 209, *221*, 269, 272, 281, *298*, 548, *574*
Rathje, W. L., 453, *454*
Reboh, R., *85*, 209, *222*
Reiter, R., 19, 24, 26, *29*, 75, *85*, 270, *299*
Resnick, C., 387, 403-4, 406-8, *421*
Rich, C., 240, 261, *264*
Rieger, C. J., 572, *574*, 592, 595, 598, 605, *619*
Riesbeck, C., 595, *619*
Rising, G., 365, *386*
Roberts, L. G., 511, 513, 535, *536*
Robinson, G. A., 74, *85*
Robinson, J. A., 58, *72*, 73, *85*, 272-3, *299*
Rosenberg, R. S., 66, *72*
Rosenfeld, A., 458, *472*
Ross, R., 59, 66, *71-2*, *185*, 273, *299*
Roussel, P., 271-2, 295, *298*
Rovner, P. D., 209, 215, *221*
Rulifson, J. F., 30, 44, *48*, 122, 130, *134*, *136*, 209, *222*
Ruth, G. R., 357, *359*
Rutovitz, D., 22
Rychener, M. D., 321, *331*
Ryder, W. R., 122, *134*

Sacerdoti, E. D., 112, 130-1, *136*, 209, *222*
Samson, E. W., 163, *168*
Sandewall, E., 14, 231-2, 235, 262, *264*
Savage, L. J., 140, 148, *150*
Saxe, A. A., 436, *454*
Schaaff, V., 439, 442, 448, *454*
Schank, R. C., 7, 171, 543, 560, *574*, 27, 576, 613-14, *619*
Schnorr, C. B., 152, *168*
Science Research Council, 173, *174*
Segre, C., 157-9, 165-6, *168*
Selfridge, O. G. 142, *150*
Shannan, G. J., 273, *299*
Shannon, C. E., 57, *72*, 154
Shaw, A. C., 525, *536*
Shaw, D. E., 106, 109, 117, *135*, 338, *359*
Shaw, J. C., 57, *71*, 120, *135*, 211, *222*
Sherwood, B. A., 392, 400, *420*
Shirai, Y., 511, 535, *536*
Shortliffe, E. H., 303, 306, 319, 326, *331*, 445, *453*
Shostak, R. E., 74, *85*
Shrobe, H., 240, 261, *264*
Sibert, E. E., 58, *71*, 273, *299*
Sickel, S., 4, *85*, 171
Simmons, 595, *619*
Simon, H. A., 57, *71*, 120, *135*, 210-11, *221-2*, 301, *329*, *331*
Sint, L., 66, *71*
Slagle, J. R., 59, *72*, *85*
Slocum, 595, *619*
Smith, B., 58, *71*
Smith, C. A. B., 140, *150*
Smith, D. C., *331*
Smith, D. H., 303, *331*
Southall, J. P. C., 517, *536*
Spowart, G., *472*
Sproul, B., 209, *222*
Stanton, R. B., 525, *536*
Steinberg, L. I., 106, 109, 117, *135*, 338, *359*
Stickel, M. E., *85*
Stroud, A., *472*
Sunnucks, A., 148, *150*

Sussman, G. J., 96, 99, 100, 112, 117-18, 122, 132, *135-6*, 209, *221*, 423, *429*, 549, *547*
Swinburne, A. C., 97
Swinehart, D., 209, *222*
Syrus, P., 99

Takeuchi, I., *93*
Tan, S. T., 175, 178, *185*, 202, *204*
Tate, A., 112, 122, *136*
Teitelman, W., 230, 236, *264*, 337, *359*
Tenczar, P., 392-7, 400, *420-1*
Tesler, L., 576, *619*
Tesler, L. G., 303, *331*
Tjio, J. H., 455, *472*
Torbijn, P. J., *93*
Trigg, C., *93*
Turing, A. N., 57, *72*, 98, *136*
Tyson, M., 7, 2, 30-34, 38, 44-5, *48*

Uhr, L., 203, *204*
Ulam, S., 11, *29*
Ullman, J. D., 56, 67, *70*

Velard, C., *85*
Viollier, D., 447, *454*
Vuillemin, J., *48*

Wagner, N. R., *93*
Waldinger, R. J., 30, 44, *48*, 6, 96, 106, 109, 112, 117-20, 122, 130-1, *134-6*, 338, *359*
Waltz, D. L., 519, 528, 531-5, *536*
Warren, 337, *359*
Warren, D. H. D., 104, 112, 121, 125, 130, *136*, 272, 295, *299*
Waterman, D. A., 312, 317, 322, 328-9, *331-2*
Watterberg, P., 157-9, 165-6, *168*
Weaver, C., 19
Weber, S., 576, *619*
Wegbreit, B., 119, *136*, 213, *221-2*, 230, *265*
Weszka, J. S., 458, *472*
Whinihan, M., *93*
Wilber, M., 122, *136*
Wilczynski, D., 122, *134*
Wilks, Y. A., 545, *574*
Winograd, T., 95, 122, *136*, 224, 241, *265*, 293, *299*, 308, 310, *332*, 446, *453*, 531, *536*, 549, *574*, 579, *619*
Winston, P. H., 510, 517-18, 528, 535, *537*
Wirth, N., 238, *265*
Wos, L., 74, *85*
Wright, E. M., 159, *168*

Yates, R. A., *85*

Zahn, C. T., 340, *359*
Zobrist, A. L., 122, *134*